Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Theses and Dissertations | Thesis Collection

1989

Integrated support for manipulation and display of 3D objects for the Command and Control Workstation of the Future

Munson, Steven Alfred.

Monterey, California. Naval Postgraduate School

DUDLEY KNOX LIBRARY

http://www.nps.edu/library

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

OUDLEY KNOX LIBRARY
[illegible] SCHOOL
MONTEREY, CALIFORNIA 93943-5002

NPS52-89-039

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

M9544

INTEGRATED SUPPORT FOR MANIPULATION AND DISPLAY OF 3D OBJECTS FOR THE COMMAND AND CONTROL WORKSTATION OF THE FUTURE

by

Steven Alfred Munson

June 1989

Thesis Advisor: Michael J. Zyda

Approved for public release; distribution is unlimited
Prepared for:

Naval Ocean Systems Center
Code 402
San Diego, CA 92152

Naval Underwater Systems Center
Combat Control Systems Division
Building 1171/1
Newport, RI 02841

T244004

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research conducted for the United States Naval Underwater Systems Center, Newport, Rhode Island and United States Naval Ocean Systems Center, San Diego, California. The work was funded by the Naval Postgraduate School.

Reproduction of all or part of this report is authorized.

This thesis is issued as a technical report with the concurrence of:

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Direct Funding
ADDRESS (City, State, and ZIP Code) Monterey, CA. 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TITLE (Include Security Classification) INTEGRATED SUPPORT FOR MANIPULATION AND DISPLAY OF 3D OBJECTS FOR THE COMMAND AND CONTROL WORKSTATION OF THE FUTURE

PERSONAL AUTHOR(S) Munson, Steven A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	June 1989	117

SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Inexpensive; Graphics; Workstation; 3D; CCWF Command and Control; Display of 3D Objects

ABSTRACT (Continue on reverse if necessary and identify by block number)
The Command and Control Workstation of the Future (CCWF) demands a variety of platforms in various configurations to accurately reflect the environment it is attempting to portray. Prior to this research, individual platforms for the CCWF and other simulations at NPS have been coded directly into each individual program, with no commonality of design or ability to readily share or modify individual platforms. The goal of this research is to develop a text-based file format for the description, modification and display of 3D objects in the CCWF and other simulations. The other primary goals of this research are the development of interactive, graphical routines to display, view, modify and then save current objects into new files; to permit conversion of text object files to binary format for compressed storage; and to develop routines that enable the CCWF and other simulations to import objects from libraries of such 3D files directly into their programs, and display them.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. Michael J. Zyda	22b. TELEPHONE (Include Area Code) 408-646-2305 22c. OFFICE SYMBOL Code 52

Approved for public release; distribution is unlimited

INTEGRATED SUPPORT FOR MANIPULATION AND DISPLAY OF 3D OBJECTS FOR THE COMMAND AND CONTROL WORKSTATION OF THE FUTURE

by

Steven Alfred Munson
Lieutenant, United States Coast Guard
B.S., United States Coast Guard Academy, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

Thesis
M9544
C.1

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

ABSTRACT

The Command and Control Workstation of the Future (CCWF) demands a variety of platforms in various configurations to accurately reflect the environment it is attempting to portray. Prior to this research, individual platforms for the CCWF and other simulations at NPS have been coded directly into each individual program, with no commonality of design or ability to readily share or modify individual platforms. The goal of this research is to develop a text-based file format for the description, modification and display of 3D objects in the CCWF and other simulations. The other primary goals of this research are the development of interactive, graphical routines to display, view, modify and then save current objects into new files; to permit conversion of text object files to binary format for compressed storage; and to develop routines that enable the CCWF and other simulations to import objects from libraries of such 3D files directly into their programs, and display them.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Captain Emil Velez, USA, who worked with me on a beginning graphics project that laid some important groundwork for this thesis, including code for object data modification. He also provided the initial programs for converting objects in the standard IRIS format to the OFF format.

I would also like to thank Captain Charles Phillips, USA, who developed the 3D objects currently used in the CCWF. These objects were converted to the OFF and used to test and debug the code for this thesis work.

I also extend my thanks to Lieutenant Michael DeHaemer, USN, who developed an initial draft of the ASCII file format, as well as the first draft of the Lex object code. His assistance in getting this project started was invaluable.

Finally, I wish to thank Professor Michael J. Zyda, whose patience, perseverance and guidance allowed this work to be completed. It is impossible to measure the difficulty of controlling the volatile mix of unbridled enthusiasm and complete inexperience this thesis student embodied. Somehow, he managed.

I. THE NEED FOR A STANDARD FILE FORMAT

A. CURRENT NPS REAL-TIME VISUAL SIMULATORS

The Naval Postgraduate School's Graphics and Video Laboratory has a long history of developing real-time visual simulators for various Department of Defense interests. One of the earliest simulators was the Fiber Optic Guided Missile (FOGM), simulated to fly over the terrain of Fort Hunter Liggett, California. It was capable of displaying a 10 kilometer by 10 kilometer area with vehicles. It was designed and implemented on the IRIS 3120 system [Ref. 1].

An extension to FOGM was the VEH simulator, which limited the terrain area drawn to the current view direction and angle. This sharply reduced the number of polygons drawn over the FOGM. VEH was ported to the IRIS 4D/70GT and included networking capability between compatible IRIS workstations [Ref. 2].

The Moving Platform Simulator (MPS), currently undergoing development, is a combination of FOGM and VEH. It takes advantage of the advanced graphics capabilities of the IRIS 4D/70GT workstation hardware. Additions to the previous work are variable terrain color schemes, collision detection, and a lighting model with a month and hour variable sun light source. Current work on MPS is being conducted to produce high resolution, (12.5 meter) terrain displays, forward artillery observation training and line of sight information displays for the FOGM/TOW missile. Additionally, new and more detailed vehicles are being developed for the simulator, including an M1 Abrahms tank.

The Commanders Display System (CDS) was the initial work toward the Command and Control Workstation of the Future (CCWF) [Ref. 3]. It was designed to

provide the contact and tactical data for simulator view programs. The CDS was written for the IRIS 2400T workstation, using Navy Tactical Display System (NTDS) fonts to display the various contacts. The CDS has not been ported to the IRIS 4D/70GT.

From the CDS, the preliminary work on the CCWF took the form of the Surface View Simulator [Ref. 4]. The Surface View Simulator was designed to display an area of operation as would be seen from the bridge of a surface ship or from a helicopter. It featured the use of resolution boundaries, high, medium and low, in order to permit simulation of areas as large as 26 nautical miles in distance on a real-time basis.

Presently, the Surface View Simulator work has been suspended in favor of the Subsurface and Periscope Views for the CCWF. This simulator allows for surface ship bridge views, as well as subsurface (submarine) viewing, multiple platforms, resolution boundaries for real-time presentation, lighting models and networking capabilities [Ref. 5].

B. IRIS GRAPHICS PORTABILITY

As can be seen from the multitude of simulator work being conducted at NPS, it is highly desirable to develop simulators that can be easily ported to upgraded models of the IRIS workstations. Development of these simulators has occurred on all levels of IRIS machines currently employed by the NPS Graphics and Video Laboratory.

At present, all platforms are coded directly into each simulator, using the IRIS graphics commands suited to the particular workstation being used, as well as having the actual physical object coordinates stored in some formal data structure within each simulator. It should be readily apparent that this format does little to enhance portability of these simulators from one machine to another, and in fact can hinder

such portability. It was the goal of this research to develop a suitable format for the storage, manipulation and display of 3D objects within the CCWF and other simulations at NPS. This format should be machine independent as far as possible, making future portability and modification a simple process.

C. THE STANDARD FILE FORMAT CONCEPT

In order to present 3D objects in a manner independent of IRIS workstation software and hardware, it was decided to develop an ASCII text format for specifying objects. Of course, this ASCII specification had to rely on current IRIS capabilities and limitations, and it is based on the IRIS 4D/70GT series graphics facilities. However, the ASCII format does not use IRIS graphics commands, and creates files of objects that can be updated and modified with any standard text editor. Furthermore, the code written to recognize and process various parts of the standard ASCII file format can be easily modified to adapt to variations in future IRIS workstation hardware and/or software.

It was understood that future objects incorporated into the simulations might well contain hundreds, if not thousands, of polygons. This is due to the fact that the IRIS graphics hardware is constantly being upgraded to display more in real-time, and also to the presence of ongoing research at NPS to produce high resolution objects for the IRIS from 3D digitizer camera imaging. Since this will inevitably be the case, ASCII storage of such object formats was considered too memory intensive. To rectify this foreseen problem, a binary format of the object files has been developed, along with conversion tools to freely convert from one form to the other.

The remainder of this work explores the file formats currently being used for 3D object storage, their usage in previewing, modifying and storing new object files, the

conversion between formats, and the integration of these routines into current and future simulations.

II. THE OBJECT FILE FORMAT - ASCII VERSION

A. DETERMINATION OF FILE TOKENS

In determining what the format of the Object File Format (OFF) should be, it was necessary to consider what might be required to provide accurate information while limiting the number of recognizable tokens to as few as possible. The ultimate objective was to create a meaningful yet concise and easy to learn file format.

The IRIS 4D/70GT workstation, for which this current format was developed, recognizes planar polygons and lines for graphical display. Furthermore, planar polygons can be described in terms of their color or material properties for the purposes of lighting and/or shading. The IRIS graphics pipeline utilizes Gouraud shading across planar polygons utilizing normals at each vertex. For polygons without vertex normals, but with a polygon normal specified, the polygon's color is uniform and determined by currently bound material characteristics, as well as those of the current lighting model and active lights. Polygons without normals of any kind are a uniform, specified color. Lines are described only by color, and are not modifiable within the current IRIS lighting models. That is, lines are a uniform, specified color regardless of current lighting models or lights defined. In fact, if lines are drawn with the IRIS in the "lighting active" mode, all lines are drawn black in color. Lines must be drawn with the lighting mode inactive to be shown in their properly set colors [Ref. 6].

Beyond these needed distinctions, it was necessary for the file format to cover the creation of lighting models, lights, and material and color definitions. Additionally, it was deemed appropriate for a clearly distinguished object title and date to be included, separate from other commenting capability. This would allow the object's title

and date to be easily recognized and, if desired, displayed in any program incorporating that file. Further, some form of file inclusion capability was deemed highly desirable, allowing the creation and easy use of "libraries" containing material, color, modeling and lighting definitions. Finally, it was desirable that the format allow as much free-format commenting as possible, so that all object files could be well self-documented.

B. HOW TOKENS RELATE TO IRIS GRAPHICS

The tokens selected to perform these file functions had to relate in some intelligible way to IRIS graphics commands, so that the mental process of creating file elements was straightforward. However, it was also desirable for tokens to utilize standard English key words as much as possible, freeing the format from direct correlation to any particular IRIS feature in order to enhance future portability.

For a complete, detailed listing of the specific token formats, please refer to Appendix A. The tokens are presented more briefly here. The tokens that do not pertain directly to the graphics portion of the file are: **title**, **date**, **origin**, **include**, and any comments. The title and date are strings designed to identify the object title and the date of creation or last file modification. The origin token allows specification of the object's origin, the position about which it is graphed, rotated, translated, etc. The default origin assumed is (0.0 0.0 0.0) if none is specified. The **include** token causes input to then be taken from the specified file, resuming after the end of that file at the current file position. Comments are designed to enhance self-documentation and clarity.

Comments follow the standard C language commenting conventions. That is, comments are contained within a pair of "/* */" markers. Unlike the C language convention, however, the limitations on token identification imposed by use of the **Lex**

generated lexical analyzer yield a maximum allowable token length of 200 characters. For this reason, all comments in the OFF should be limited to at most two lines. Consecutive lines of comments are not a problem. Figure 2.1 shows a sample of the **title, date, origin, include** and comment tokens.

```
/*  My first sample object file */

/* the object title is the following */
title  "My First Carrier"

/* file created */
date   15 May 1989

/* origin is other than standard 0 0 0 */
origin
0.0  5.0  -10.0

/* be sure to read materials defined for the carrier */
include  "Carrier.materials"
```

Figure 2.1 Sample Title, Date, Origin, Include and Comments

Lines are defined by the **defline** token. Each **defline** is followed by an integer number specifying the number of line segment vertices, and then x y z triples of floating point vertex coordinates. These coordinates are graphed from within an IRIS 'bgnline() .... endline()' sequence of commands. Figure 2.2 shows sample line definitions and associated data.

Planar polygons are obliged to have either a single polygon normal or vertex normals. The former case is referred to as a "polygon" in the OFF, and the latter is a "surface." Each is handled a bit differently.

Polygons are identified by a **defpoly** token. Following this is the x y z triple showing the normalized (unit length) polygon normal vector. This, in turn, is followed by the number of vertices, then x y z vertex triples. The vertices are graphically

displayed by an IRIS 'bgnpolygon() .... endpolygon()' sequence, utilizing the single polygon normal for lighting calculations. Figure 2.3 shows sample polygon definitions.

```
/* lines for carrier antennas */

defline
4
-86.000000 26.000000 39.000000
-85.000000 40.000000 38.000000
-83.000000 40.000000 38.000000
-82.000000 26.000000 39.000000

defline
4
-86.000000 26.000000 35.000000
-85.000000 40.000000 36.000000
-83.000000 40.000000 36.000000
-82.000000 26.000000 35.000000
```

Figure 2.2 Sample Defline Tokens and Data

Surfaces are identified by a **defsurface** token. This is followed by the number of vertices in the surface, then by x y z i j k sextuples of vertices and normalized (unit length) vertex normals. The vertices are again graphically displayed from within an IRIS 'bgnpolygon() .... endpolygon()' sequence, but with normals specified at each vertex. Figure 2.3 shows sample surface definitions.

Lighting models are identified by the **deflmodel** token. The file can then modify any of the IRIS lighting model features, including ambient background color, lighting attenuation calculations, and whether the viewer position is local or not. Attributes of the lighting model not specified are set to current IRIS default values. The end of the model definition is determined by the use of the **defend** token. Figure 2.4 shows a sample lighting model definition.

```
/* some sample polygons, complete with normals */

defpoly
-0.422903 -0.640762 0.640762
3
0.000000 0.000000 0.000000
-0.500000 0.220000 -0.110000
-0.500000 0.110000 -0.220000

defpoly
-0.402739 0.000000 0.915315
3
0.000000 0.000000 0.000000
-0.500000 0.110000 -0.220000
-0.500000 -0.110000 -0.220000

/* here are some cube face surfaces */
defsurface
4
20.000000 20.000000 20.000000 0.333333 0.333333 0.333333
20.000000 20.000000 -20.000000 0.333333 0.333333 -0.333333
20.000000 -20.000000 -20.000000 0.333333 -0.333333 -0.333333
20.000000 -20.000000 20.000000 0.333333 -0.333333 0.333333

defsurface
4
20.000000 20.000000 20.000000 0.333333 0.333333 0.333333
20.000000 20.000000 -20.000000 0.333333 0.333333 -0.333333
-20.000000 20.000000 -20.000000 -0.333333 0.333333 -0.333333
-20.000000 20.000000 20.000000 -0.333333 0.333333 0.333333
```

Figure 2.3 Sample Polygon and Surface Definitions

```
/* sample light model showing all attributes */

deflmodel
ambient 0.2 0.2 0.2
localviewer 1.0
attenuation 1.0 0.5
defend
```

Figure 2.4 Sample Lighting Model Definition

Lights are identified by the **deflight** token. The IRIS light attributes can be modified from their default values. These attributes include the light's ambient contribution, its color and its position. The position is specified by four floating point numbers. The first three are an x, y and z coordinate in three dimensional space. The fourth value indicates whether the light is at infinite distance or is a local light. In the former case, the position is actually a direction to the infinite light source. In the latter, it is an actual light position. Again, **defend** signals the end of the current definition. Figure 2.5 shows a sample light definition.

```
/* sample light definition of a red light */

deflight   redlight
ambient 0.1 0.5 0.5
lcolor 0.0 1.0 0.0
position 13.0 35.0 10.0 1.0
defend
```

Figure 2.5 Sample Light Definition

Material definitions for polygon material composition are identified by the **defmaterial** token. The material must be named for future reference within the object file. IRIS material properties can be modified from their default values. These properties include material emission color, ambient color, diffuse color, specular highlighting

color, shininess and the material's alpha value, used in IRIS lighting calculations. Once again, **defend** signals the end of material attributes. A sample material definition is found in Figure 2.6.

Colors used in line drawing are identified by the **defcolor** token. The color must be named for future reference. This is followed by three floating point values, between 0.0 and 1.0, which correspond, respectively, to the red, green and blue component of the color. Colors are activated by use of the IRIS c3f() call. A sample color definition is found in Figure 2.6.

```
/* examples of defining and setting */
/* materials and colors */

/* first, a material */
defmaterial    gold
  ambient 0.4 0.2 0.0
  diffuse 0.9 0.5 0.0
  specular 0.7 0.7 0.0
  shininess 10.0
  defend

setmaterial gold

/* now a color */

/*  color name quoted because it is contains a blank space */
defcolor    "Antenna Color"
0.0 0.0 0.0

setcolor  "Antenna Color"
```

Figure 2.6 Sample Material and Color Tokens

Materials are set, or activated, for all future surfaces and polygons by use of the **setmaterial** token. This token is followed by the name of the material to be used. Thereafter, until another **setmaterial** is invoked, all polygons and surfaces will be of this material type. A sample use of **setmaterial** is shown in Figure 2.6.

Colors for lines are likewise activated by use of the **setcolor** token, followed by the name of the color to be used. All future lines are drawn in that color until another **setcolor** token is invoked. A sample use of **setcolor** is shown in Figure 2.6.

For the purposes of the **preview** program tool, and simulator use in general, there is no ability to specify planar polygons in the OFF without normals, polygon or vertex. This is because all simulators currently under development at NPS avail themselves of the IRIS lighting capabilities, and it is unforeseeable that future simulator research will not follow this path. Therefore, the **setcolor** token affects only the color of all further lines, defined by the **defline** token, in the object file.

As Appendix A illustrates, the names of properties used for lights, lighting models and materials corresponds directly to the aspect of the IRIS feature which they represent. However, their exact format can be easily changed in the future without loss of generality and readability in the OFF.

C. EDITABILTY OF THE ASCII FORMAT

The OFF allows complete control of every aspect of the object's appearance from within the file. The lighting model, lights used, material and color definitions, use of lines, as well as use of polygon or vertex normals can all be controlled. At the same time, the format is easy to read and understand. It can be learned in a few short minutes. Its free use of commenting makes it possible, though not automatic, to write well-documented object files.

The fact that the standard format is an ASCII file means that any object file can be modified from within any standard text editor with which the user may be familiar, including vi, EMACS, Wordperfect, Appleworks, etc. While it is possible that improvements in future IRIS models may modify the properties of lighting models, lights or material definitions, the OFF can easily be adapted to such changes. Lines,

polygons and surfaces will doubtless remain unchanged in their format for the foreseeable future. Similarly, the process of setting materials or colors for use will remain unchanged in the format. The graphics code behind the format will simply change to seize upon new IRIS improvements.

III. THE OBJECT FILE FORMAT - BINARY

A. ARRANGEMENT OF THE BINARY FILE

As a user of a binary data file should know, in order to read the file successfully it is *imperative* that the precise format of the file be known and followed, or meaningful input from the file is impossible. Each of the tokens from the ASCII file format have been assigned to an integer value. These tokens are thus stored in the binary format as integers, and these integer tokens are used to determine the format of the following data related to each token. There are currently 24 recognizable tokens and associated integer values.

Each token has a required group of data, in a specific format, that must follow it. The precise formatting is spelled out in detail in Appendix B, and explained more briefly in section B of this chapter. The order of tokens and their associated data within the file is irrelevant, except as pertains to **setcolor** and/or **setmaterial** tokens, which control polygon and line colors for all further lines, polygons and surfaces defined in the file.

B. TOKEN GROUPINGS AND REQUIRED ELEMENTS

Each of the allowable tokens is required to be followed by a specified grouping of data in a fixed format. These tokens and formats are as outlined in the following paragraphs.

The **title** token is followed by an integer number *n* showing the length of the title (the number of characters in it), then by *n* characters (see Figure 3.1).

The **date** token is followed by an integer number *n* showing the length of the date (the number of characters in it), then by *n* characters (see Figure 3.1).

The **comment** token is followed by an integer number *n* showing the length of the comment (the number of characters in it), then by *n* characters (see Figure 3.1).

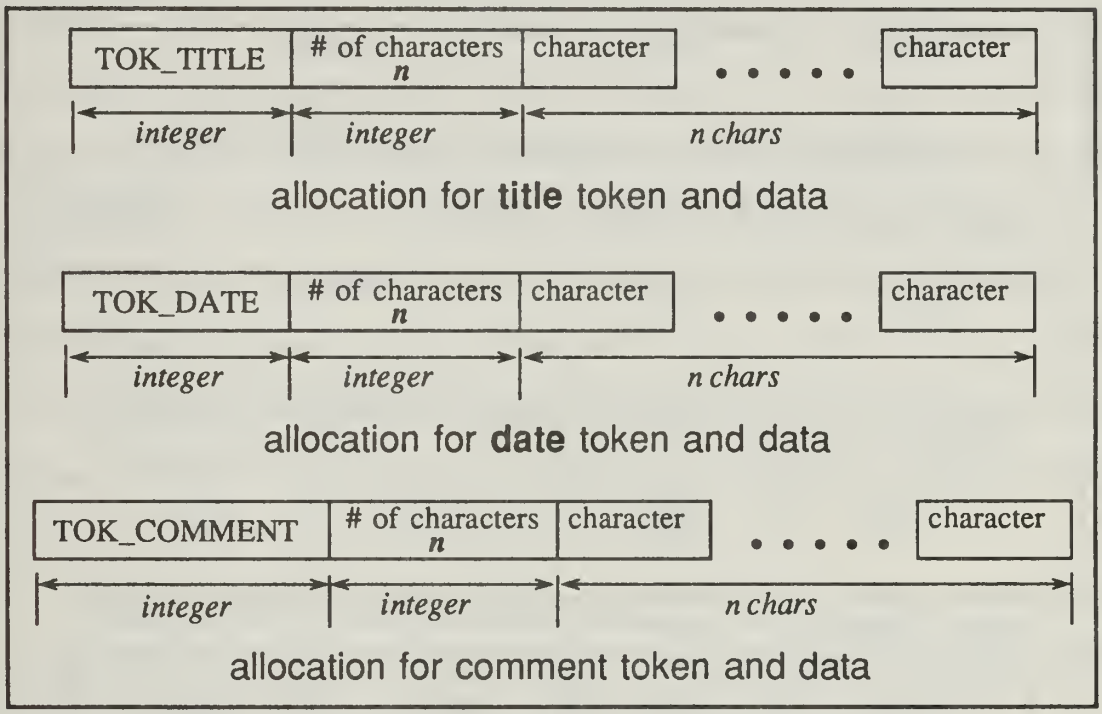

Figure 3.1 Binary Storage for Title, Date and Comment

The light definition token, **deflight**, is followed by an integer *n* indicating the length of the light's name. This is followed by the *n* characters of the name. Thereafter until the **defend** token is encountered, the following tokens and their associated data are allowed, in any order: one or more **comment** tokens, in the format described above; an **lcolor** token, followed by three floating point numbers; an **ambient** token, followed by three floating point numbers; a **position** token, followed by four floating point numbers. Figure 3.2 illustrates this storage.

The lighting model definition token is **deflmodel**. Until the **defend** token is encountered, the following tokens and their associated data are allowed, in any order:

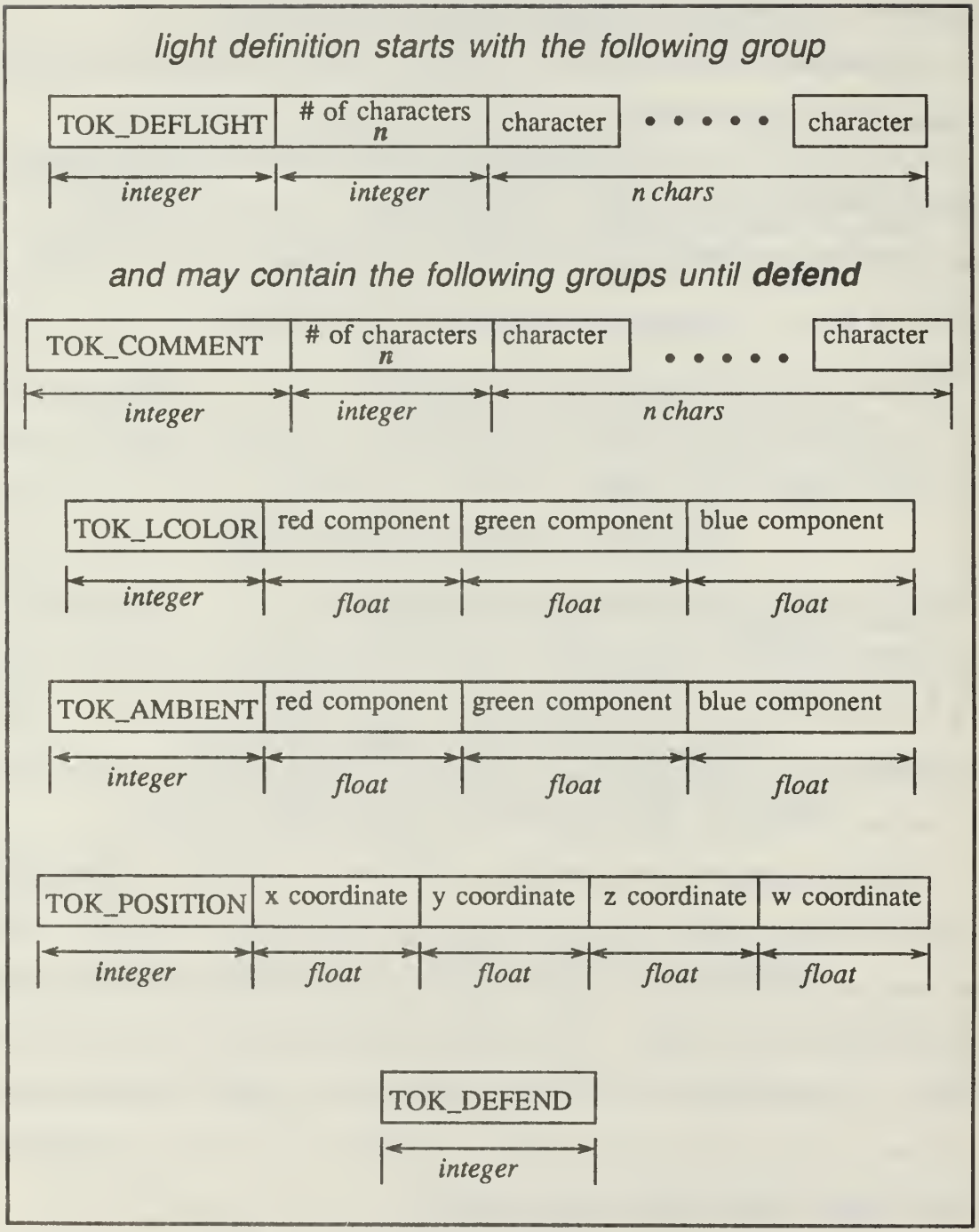

Figure 3.2 Storage of a Light Definition

one or more **comment** tokens, in the format described previously; a **localviewer** token, followed by a floating point number; an **ambient** token, followed by three floating point numbers; an **attenuation** token, followed by two floating point numbers. Figure 3.3 illustrates this storage.

The material definition token, **defmaterial**, is followed by an integer *n* indicating the length of the material's name. This is followed by the *n* characters of the name. Thereafter, until the **defend** token is encountered the following tokens and their associated data are allowed, in any order: one or more **comment** tokens, in the format described previously; an **emission** token, followed by three floating point numbers; an **ambient** token, followed by three floating point numbers; a **diffuse** token, followed by three floating point numbers; a **specular** token followed by three floating point numbers; a **shininess** token followed by one floating point number; an **alpha** token, followed by one floating point number. Figure 3.4 illustrates this storage.

The color definition token, **defcolor**, is followed by an integer *n* indicating the length of the color's name. This is followed by the *n* characters of the name. Finally, three floating point numbers conclude this token group (see Figure 3.5).

The token **origin** is followed by three floating point numbers, corresponding to the x, y and z values of the specified object origin (see Figure 3.5).

The **setmaterial** token is followed by an integer *n* indicating the length of the material's name. This is followed by the *n* characters of the name (see Figure 3.5).

The **setcolor** token is followed by an integer *n* indicating the length of the color's name. This is followed by the *n* characters of the name (see Figure 3.5).

The token to define a line is **defline**. This is followed by an integer *n* indicating the number of vertices to be connected in the line. Finally, *3n* floating point numbers

follow, giving the x, y and z coordinates of each vertex in the line. Figure 3.6 illustrates this storage.

The token to define a polygon is **defpoly**. This is immediately followed by three floating point numbers, providing the polygon surface normal x, y and z coordinates. This is followed by an integer *n* indicating the number of vertices to be connected in the polygon. Finally, *3n* floating point numbers follow, giving the x, y and z coordinates of each vertex in the polygon (see Figure 3.6).

The token to define a surface is **defsurface**. This is followed by an integer *n* indicating the number of vertices to be connected in the surface. Finally, *6n* floating point numbers follow, giving the x, y and z coordinates of each vertex in the surface followed immediately by the i, j and k vertex normal coordinates (see Figure 3.6).

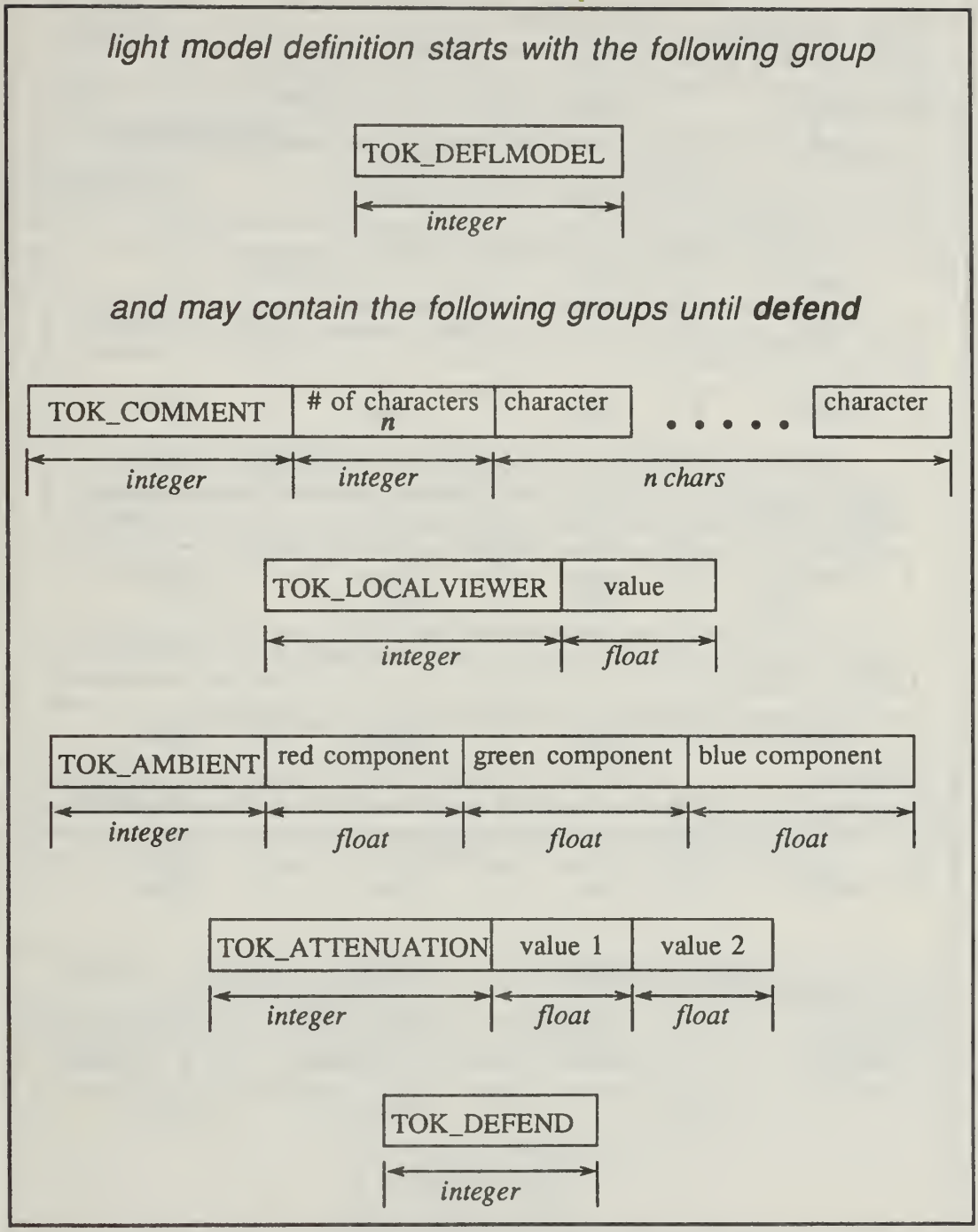

Figure 3.3 Storage of a Lighting Model Definition

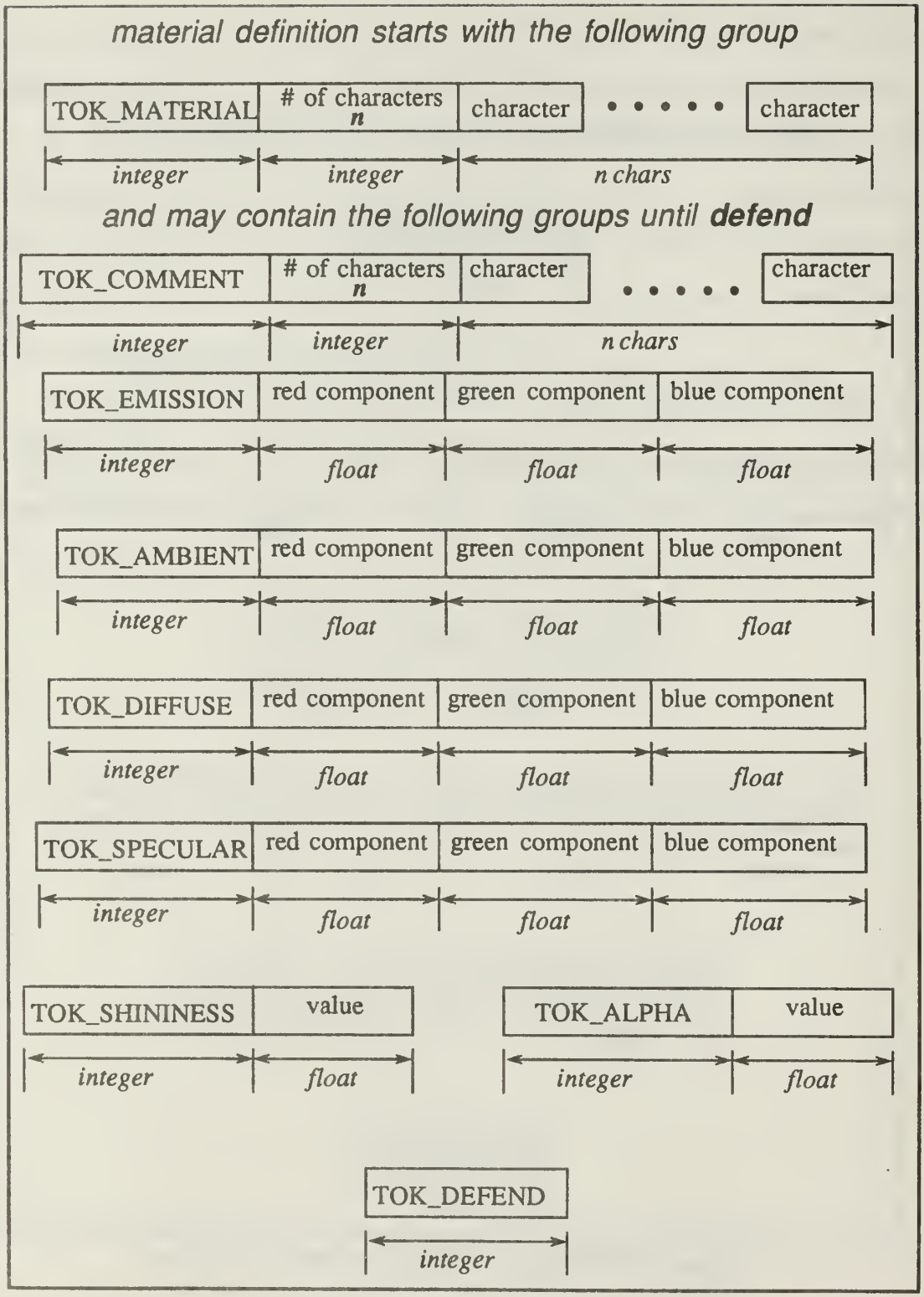

Figure 3.4 Material Definition Storage

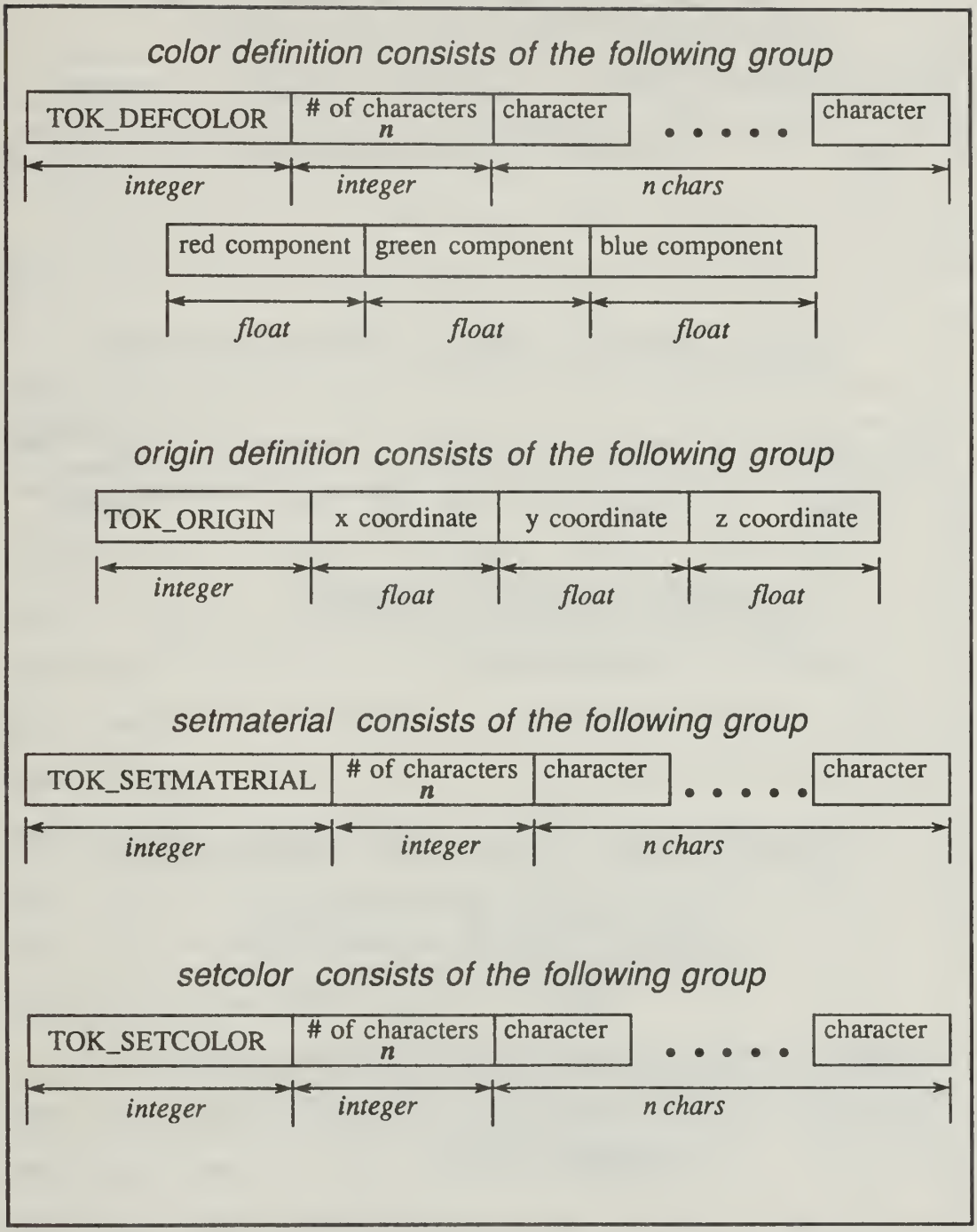

Figure 3.5 Storage of Defcolor, Origin, Setmaterial, Setcolor

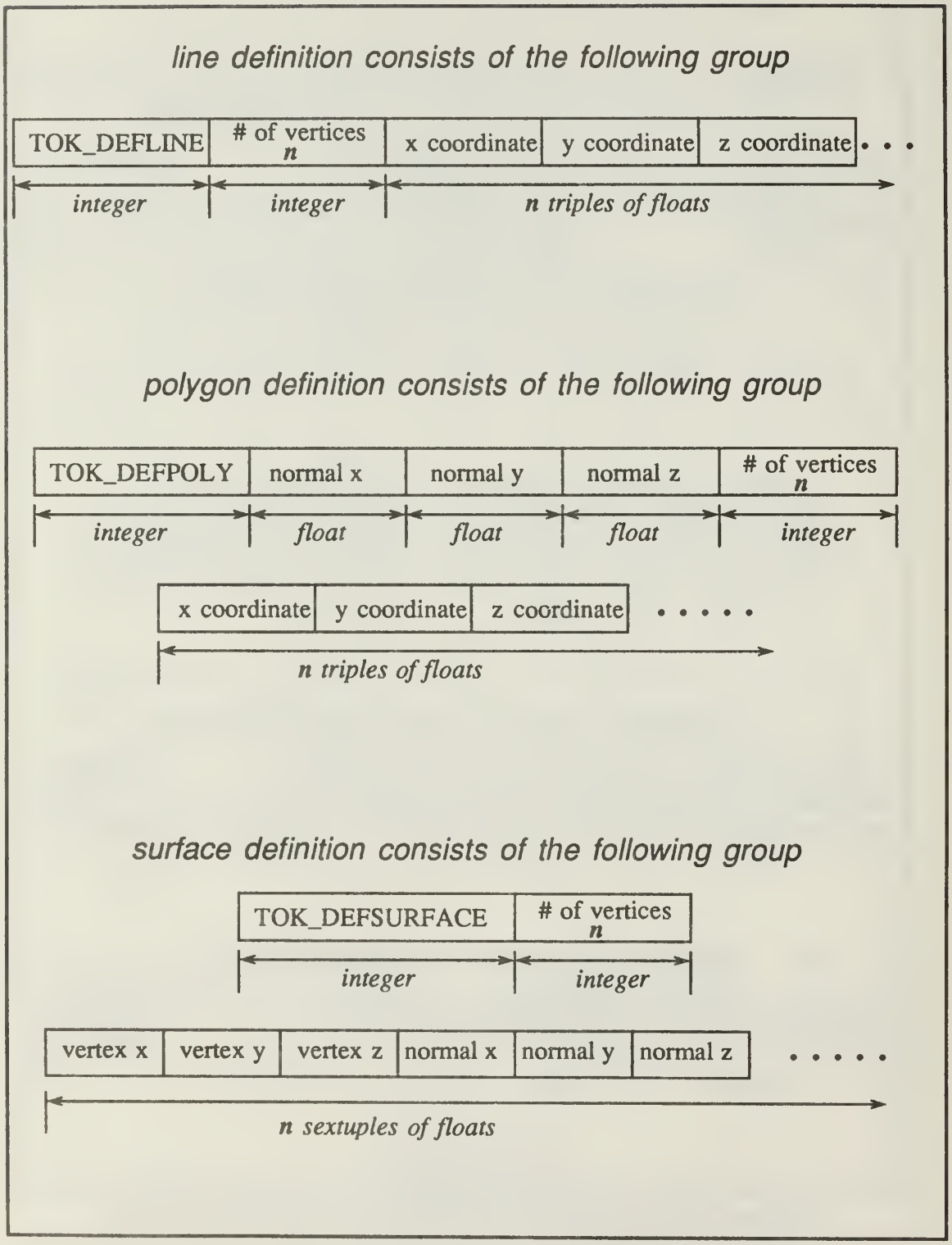

Figure 3.6 Line, Polygon and Surface Storage

IV. THE DYNAMIC OBJECT STRUCTURE

A. THE PREMISE OF THE STRUCTURE

The top-level goal of this research was to permit 3D objects of any size and complexity to be read into some formal data structure, and then displayed in any simulator or other program in use. Because each object varies radically in the number of lines, polygons and surfaces, as well as the materials and colors used, a dynamically allocated structure is necessary.

Since interactive previewing and modification tools are desired, traversing through the object's structure is a requirement. Further, to ensure object structure robustness, that is to say, to allow future additions and deletions of object elements beyond those currently in use, some form of generic structure is ideally suited to be used here. Therefore, it was decided that each object should be stored in a dynamically allocated structure, where the object was "stored" in one *object header* structure that contained pertinent information such as title, date, object origin, maximum and minimum values, etc., and which contained pointers to the heads and tails of doubly-linked lists of the object's elements (lines, lights, materials, polygons, etc.). Figure 4.1 shows the code defining the *object header* structure.

It is impossible to display an object by merely traversing lists of independent elements. That is, by traversing the list of polygons or surfaces, it is impossible to tell which material had been selected to use for individual elements. Hence the header structure also contains a head and tail pointer for a *draw list*. It can contain up to five different types of *double nodes*. These include material, color, line, polygon or surface nodes. Material nodes indicate a change of material to be applied to all future

```
/* define the OBJECT_HEADER structure type */
typedef struct object_header OBJECT_HEADER;

struct object_header        /* main head node for the project */
{
    char      *title;        /* object title */
    char      *date;         /* date of object file */
    float     origin[3];     /* object origin */
    float     maxx,          /* max and min x,y,z  */
              minx,          /* coordinates of the polygons */
              maxy,          /* or surfaces of the object. */
              miny,
              maxz,
              minz;
    /* ptrs to head and tail of list of light definitions */
    double_ptr      head_lightdefs;
    double_ptr      tail_lightdefs;
    /* ptrs to head and tail of list of lighting model definitions */
    double_ptr      head_modeldefs;
    double_ptr      tail_modeldefs;
    /* ptrs to head and tail of list of material definitions */
    double_ptr      head_materialdefs;
    double_ptr      tail_materialdefs;
    /* ptrs to head and tail of list of color definitions */
    double_ptr      head_colordefs;
    double_ptr      tail_colordefs;
    /* ptrs to head and tail of list of line definitions */
    double_ptr      head_linedefs;
    double_ptr      tail_linedefs;
    /* ptrs to head and tail of list of polygon definitions */
    double_ptr      head_polygondefs;
    double_ptr      tail_polygondefs;
    /* ptrs to head and tail of list of surface definitions */
    double_ptr      head_surfacedefs;
    double_ptr      tail_surfacedefs;
    /* ptrs to head and tail of list of draw list */
    double_ptr      head_drawlist;
    double_ptr      tail_drawlist;
};
```

Figure 4.1 Object Header Definition Code

polygons and surfaces, until another material node is encountered. Color nodes indicate a change to be applied to succeeding lines until another color node is encountered. Lines, polygons and surfaces are object elements to be displayed. Figure 4.2 illustrates the draw list concept.

B. GENERIC NODES AND POINTERS

In order to permit the addition of various structure types to the *draw list*, as well as to maintain robustness and standardization between the various lists of elements in the dynamic object structure, the character pointer was utilized. That is, a type definition was made so that a *generic pointer* is actually a character pointer. These *generic* pointers can then be coerced to point to any structure desired.

Each node in any given doubly-linked list is a *double node*. It contains two *double pointers*, one to its predecessor node and one to its successor node. A *double pointer* has been type defined to be a pointer to a *double node* structure. There is one *generic pointer* in each *double node* that points to a dynamically allocated structure where the data is stored for that node. It also contains an integer *dtype* field indicating what type of data is being pointed to by the *generic pointer*. The list's head and tail *double pointers* are maintained in the object's *header* structure. Figure 4.3 provides the code defining the node structure, pointers and macros.

The result of using double nodes and generic pointers is several fold. First, appending to or inserting in any doubly-linked list of elements can be accomplished by one set of primitive append and insert routines. Second, any double node can point to any data structure the user desires. At present, there are seven such data storage structures, one each for lights, lighting models, colors, materials, lines, polygons and surfaces. Finally, any new structures corresponding to new elements added to the

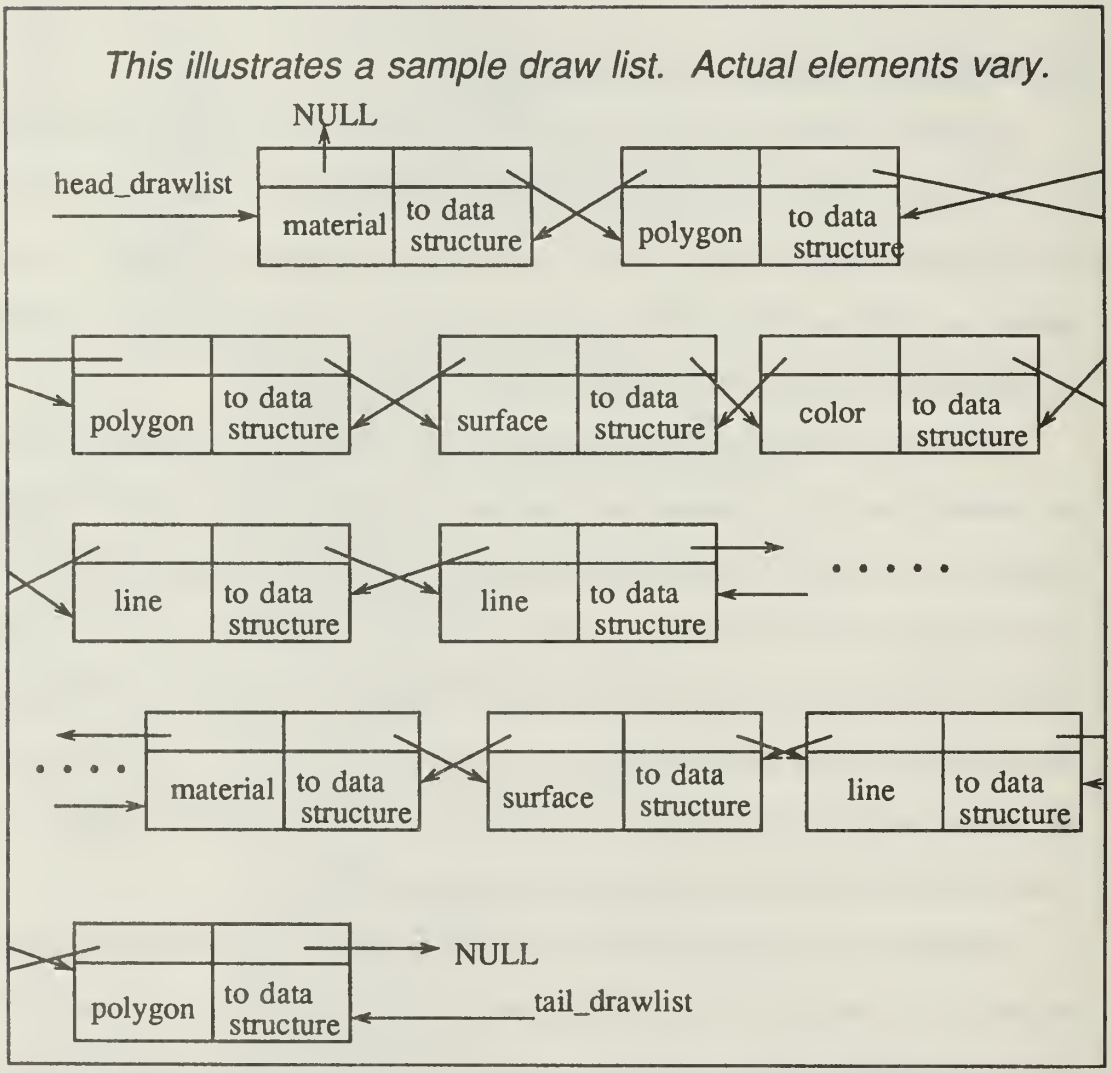

Figure 4.2 Object Draw List Illustrated

object file format can be easily accommodated using the present node and pointer structures and list handling routines. Figure 4.4 illustrates the use of *double nodes*.

```
/* define what a double node and a pointer to a double node are */
typedef struct node double_node, *double_ptr;

struct node          /* generic node in doubly-linked list */
{
  generic_ptr  data;     /* pointer to any data structure */
  int          dtype;    /* data type indicated by defined tokens */
  double_ptr   prev;     /* pointer to previous node */
  double_ptr   next;     /* pointer to next node */
};  /* end node */

/* definition MACROS for use in list handling routines */
#define DATA(L)     ((L)->data)
#define DATATYPE(L) ((L)->dtype)
#define NEXT(L)     ((L)->next)
#define PREV(L)     ((L)->prev)
```

Figure 4.3 Node Definition Code

C. ORDERING OF ELEMENTS WITHIN THE LISTS

Within each list of elements (lights, materials, surfaces, etc.) for the object, it was necessary to determine in which order to add new nodes as data was read from the object file. In general, new nodes are added at the end of each list of elements, and the tail pointer in the *header* structure updated. The *draw list* has elements added to its end as well, except in the **preview** tool program, discussed in chapter VI. Lines, polygons, surfaces, materials and colors are inserted at the tail of the *draw list* as they are encountered. Figure 4.5 illustrates this insertion.

D. LIST HANDLING ROUTINES

The routines for handling each of the eight lists in the header structure are the same. That is, one set of routines handles them all, since they all consist of *double*

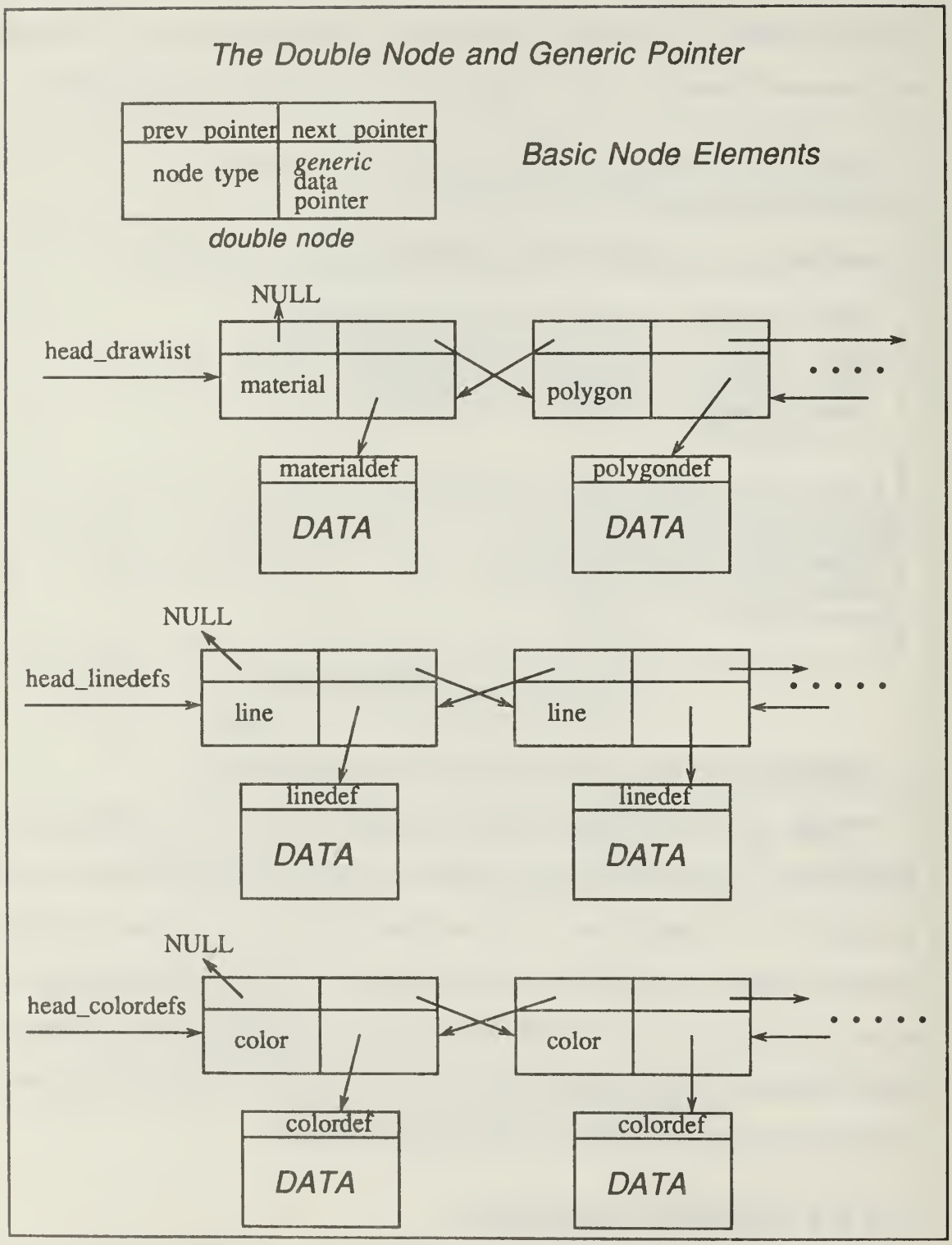

Figure 4.4 Double Node and Generic Pointer

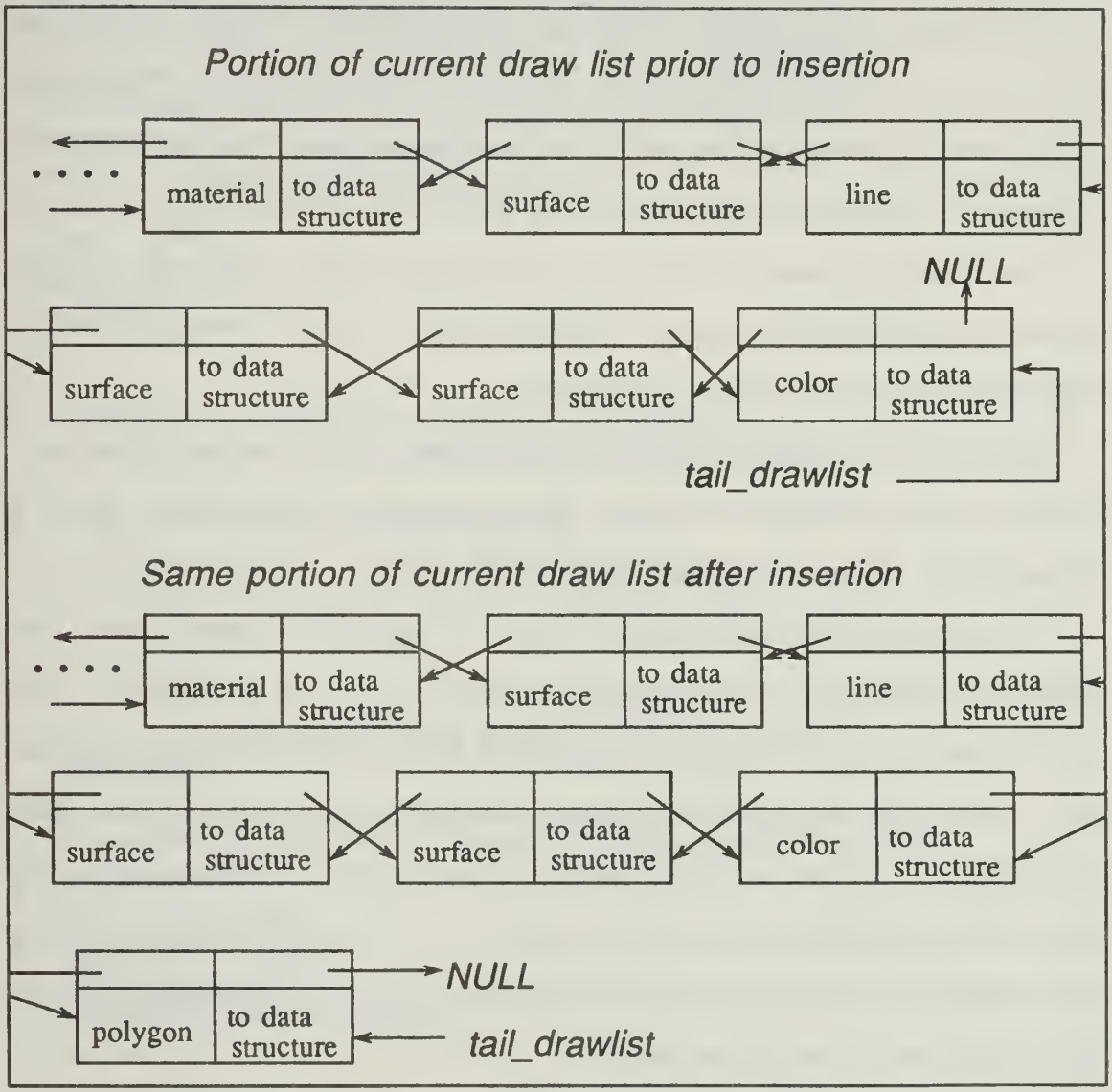

Figure 4.5 Node Insertion Illustrated

nodes. The basic routines accomplish the following tasks: allocating and assigning a new *double node*, checking a list to see if it is empty, inserting a new node in front of an existing node, and appending a node after an existing node.

The allocate_double_node() routine takes as its arguments a pointer *ptr* to a *double node*, a *generic pointer* to the data for the new node, and a node type. This routine then dynamically allocates a new *double node* and sets the value of *ptr* to this allocated memory. It then sets the *dtype* integer flag field of the new *double node* to the value of the node type passed in, sets the new node's data pointer to the *generic pointer* passed in, and sets the new node's predecessor and successor pointers to NULL. Figure 4.6 is the code for *double node* allocation.

The routine to check whether a list is empty merely checks the head of the list to see if the value of that pointer is NULL. If it is, the function returns TRUE. If not, it returns FALSE. Figure 4.7 contains the code for this routine.

The double_insert() routine takes as one of its arguments a *double pointer, ptr*, pointing to the node in the list before which the new node is to be inserted. It also takes a *generic pointer* to the data structure of the new node, and an integer data type. It then calls the routine to allocate a new *double node*, which allocates space and assigns the data type and *generic pointer* of the new node. Finally, it updates the new node's predecessor and successor pointers, as well as the predecessor pointer of the insertion point (*ptr*), and the successor pointer of the former predecessor of *ptr*. Figure 4.8 shows the code for this routine.

The double_append() routine takes as one of its arguments a *double pointer, ptr*, pointing to the node in the list after which the new node is to be inserted. It also takes a *generic pointer* to the data structure of the new node, and an integer data type. It then calls the routine to allocate a new *double node*, which allocates space and assigns the data type and *generic pointer* of the new node. Finally, it updates the

```
/* this routine allocates and establishes a new node for a generic */
/* doubly-linked list, where the pointer to the node is returned via */
/* "ptr" and the data pointed to or linked via the node is provided */
/* by the argument "data", a generic pointer to the data structure */
/* containing the data for that node */
int allocate_double_node(ptr,data,nodetype)

double_ptr *ptr;
generic_ptr data;
int nodetype;

{
 /* allocate space for new node in memory, call it item */
 double_ptr item = (double_ptr)malloc(sizeof(double_node));

 /* check to make sure malloc call worked! */
 if (item == NULL) {
   printf("MALLOC FAILURE -- MAJOR PROBLEMS!\n");
   return(FALSE);
 }   /* end if */

 /*  set pointer of the node to node "item" for return to the caller */
 *ptr = item;

 /* using MACROS defined in "filespec.h", assign data and pointers in */
 /*   the new node */
 DATA(item) = data;
 NEXT(item) = NULL;
 PREV(item) = NULL;
 DATATYPE(item) = nodetype;
 return(TRUE);

}  /* end allocate double node */
```

Figure 4.6 Allocate Double Node Code

```
/* this routine checks to see if a doubly-linked list is empty */
int empty_double_list(L)

double_ptr L;
{
 return (L == NULL) ? TRUE : FALSE;
}  /* end empty double list */
```

Figure 4.7 Code to Check for Empty List

new node's predecessor and successor pointers, as well as the successor pointer of the insertion point (*ptr*), and the predecessor pointer of the former successor of *ptr*. Figure 4.9 contains the code for this routine.

```
/* this routine inserts a new node into a doubly-linked list in front of */
/* the position pointed to by "ptr", NOT necessarily the start of the list */
int double_insert(ptr,data,type)

double_ptr *ptr;   /* pointer to insertion/starting point in list */
generic_ptr data;  /* pointer to data structure */
int type;          /* integer flag indicating data structure type */
{
 double_ptr newnode;   /* pointer to new node to create and insert */

 /* allocate new space, and return error msg if failed */
 if (!allocate_double_node(&newnode,data,type)) {
   printf("While trying to insert, allocation for new node failed.\n");
   printf("You have MAJOR MEMORY problems.  Sorry. \n");
   return(FALSE);
 }  /* end if */

 /* otherwise, go ahead and insert this new node */
 /* first case, if list is currently empty */
 if (empty_double_list(*ptr) == TRUE) {
   PREV(newnode) = NEXT(newnode) = NULL;

 /* second case, non-empty list */
 } else {

   /* new node's next pointer is set to old start */
   NEXT(newnode) = *ptr;

   /* new node's previous pointer is set to old start's previous pointer */
   PREV(newnode) = PREV(*ptr);

   /* old start's previous pointer is now new node */
   PREV(*ptr) = newnode;

   /* if old start had a previous node, set it's next pointer */
   /* to the new node */
   if (PREV(newnode) != NULL) {
      NEXT(PREV(newnode)) = newnode;
   }  /* end if */

 }   /* end if then else */

 /* set old start to new node */
 *ptr = newnode;
 return(TRUE);   /* success */

}  /* end double insert */
```

Figure 4.8 Double Insert Routine Code

```
/* this routine adds a new node to a doubly-linked list AFTER the position */
/* pointed to by "ptr", NOT necessarily the end of the list */
int double_append(ptr,data,type)

double_ptr *ptr;   /* pointer to insertion point in list */
generic_ptr data;  /* pointer to data structure holding data for node */
int type;          /* flag indicating data structure type */
{
 double_ptr newnode;

 /* allocate new space, and return error msg if failed */
 if (!allocate_double_node(&newnode,data,type)) {
   printf("While trying to insert, allocation for new node failed.\n");
   printf("You have MAJOR MEMORY problems.  Sorry. \n");
   return(FALSE);
 }  /* end if */

 /* otherwise, go ahead and insert this new node */
 /* first case, if list is currently empty */
 if (empty_double_list(*ptr) == TRUE) {
   PREV(newnode) = NEXT(newnode) = NULL;

 /* second case, non-empty list */
 } else {

   /* new node's next pointer is set to old start's next pointer */
   NEXT(newnode) = NEXT(*ptr);

   /* new node's previous pointer is set to old start */
   PREV(newnode) = *ptr;

   /* old start's next pointer is now new node */
   NEXT(*ptr) = newnode;

   /* if old start had a next node, set it's previous pointer */
   /* to the new node */
   if (NEXT(newnode) != NULL) {
      PREV(NEXT(newnode)) = newnode;
   }  /* end if */

 }   /* end if then else */

 /* set old start to new node */
 *ptr = newnode;
 return(TRUE);   /* success */

}  /* end double append */
```

Figure 4.9 Double Append Routine Code

V. THE FILE CONVERSION TOOL

The file conversion program tool is called **fileconvert**. It is located in the *off/tools/convert* directory, and is invoked by entering *fileconvert* followed by a file name. For example, to convert the ASCII object file named "Carrier.mine" to its binary equivalent, the invocation would be:

fileconvert Carrier.mine

Fileconvert determines for itself whether the specified file is in ASCII or binary OFF format, then invokes the proper conversion routines to perform the conversion to the other format. No special names or suffixes are required for any object file.

A. CONVERTING ASCII TO BINARY

The routines to convert an object file in the standard ASCII format are located in files **converttobinary.c** and **writebinary.c**. The first file contains the routine to coordinate writing to the output binary file based upon the token read from the input ASCII file. The second file contains the actual individual token writing routines invoked from **converttobinary.c**.

The code in **converttobinary.c** first creates and then opens a binary file with the same name as the input ASCII file, but with the tag ".bin" appended to it. Next, the input ASCII file is opened. After the input and output files are thus opened, a control loop is entered that executes until an end of input condition is reached in the ASCII file.

Based upon the next token read from the input ASCII file, the switch statement in the control loop invokes a routine to read the appropriate data from the input file and write it to the binary file. Each of the actual routines are found in the file

writebinary.c. Thus the routine is, essentially, one while loop testing for the end of input condition, with a switch statement nested inside.

When the end of input condition is reached, the main loop exits. The binary file, if successfully opened, is then closed. Finally, a message is printed to the screen showing the name of the new binary file just created.

B. CONVERTING BINARY TO ASCII

Similar to the ASCII to binary conversion, the routines to perform the conversion from binary to ASCII are located in the files **converttoascii.c** and **writeascii.c**. The former contains the main control routine, while the latter contains the individual routines invoked to read and write specific token groupings.

The code in **converttoascii.c** creates and opens an ASCII output file with the same name as the specified binary input file, but with the tag ".ascii" appended to it. Next, it opens for input the specified binary file. It then enters a while loop, which checks for a successful read of the next token from the input file. Inside the while loop is a switch statement, which evaluates the token read and invokes the appropriate routine to read the rest of the data for that token, then write the token and data to the output ASCII file.

Once the end of file condition is reached in the input binary file, the while loop exits. The input and output files are then closed, and a message is printed to the screen indicating the name of the new ASCII file just created.

C. A SAMPLE BINARY CONVERSION DISCUSSED

To illustrate how a binary conversion is accomplished, consider an ASCII file containing the **title** "My First Object". The format of such a title in an object file would appear similar to: **title** "My First Object".

As the input file tokens are read in the control loop of **converttobinary.c**, the token **title** would be identified and the variable *token* in that routine would be set to TOK_TITLE. This causes the switch statement to issue a call to write_binary_title(), a routine found in **writebinary.c**. The write_binary_title() routine expects that, following a title token, the next token returned from the input ASCII file will be an identifier token. If, in fact, such a token is returned, its value has been copied into the global variable *id*. The write_binary_title() routine then writes to the output binary file the TOK_TITLE integer token, then the length of the identifier (title string), then the string itself. This completes the conversion of an ASCII title to its binary equivalent. Figure 5.1 shows the code used to accomplish this.

D. A SAMPLE ASCII CONVERSION DISCUSSED

To illustrate how an ASCII conversion is accomplished, consider a binary file containing the **date** "23 Jan 1989". The format of such a date in an object binary file would be: TOK_DATE (integer), date size (integer), date string (char * date size).

As the input file tokens are read in the control loop of **converttoascii.c**, the token **date** would be identified and the variable *token* in that routine would be set to TOK_DATE. This causes the switch statement to issue a call to write_ascii_date(), a routine found in **writeascii.c**. The write_ascii_date() routine expects that, following a date token, the next value in the input binary file will be an integer. Using this integer as a date size guide, memory is allocated for size+1 characters. The date string is then read from the binary file into this allocated space. An 'end of string' character is placed at the size+1 position. Finally, the write_binary_date() routine writes to the output ASCII file the string "date ", followed on the next line by the date

```
/* this routine writes the ASCII file title to the binary file */
write_binary_title(bf)

int bf;

{
 int   nexttoken;  /* next token in the input stream */
 int   firsttoken = TOK_TITLE;
 int   titlesize;

 /* get next token from input file */
 nexttoken=yylex();

 /* should be an ID -- a "string" */
 if (nexttoken==TOK_ID) {
   titlesize = strlen(id);
   write(bf,&firsttoken,sizeof(int));
   write(bf,&titlesize,sizeof(int));
   write(bf,id,sizeof(char)*titlesize);

 /* if not, error! */
 } else {
   printf("ERROR -- no identifier after key token TITLE \n");
   printf("No title written to binary file\n");
 }
 return(1);
}  /* end write binary title */
```

Figure 5.1 Code to Convert an ASCII Title to Binary

string and a new line character. This completes the conversion of a binary date to its ASCII equivalent. Figure 5.2 shows the code used for this operation.

```
/* this routine reads and copies a date to the ASCII file */
write_ascii_date(bf,outfile)

FILE *outfile;
int bf;

{
  int datesize;  /* size of date string */
  char *date;

  read(bf,&datesize,sizeof(int));
  date = (char *)malloc(sizeof(char)*(datesize+1));
  read(bf,date,sizeof(char)*datesize);
  date[datesize] = '\0';
  fprintf(outfile,"\ndate\n");
  fprintf(outfile,"%s\n",date);

}  /* end of write_ascii_date */
```

Figure 5.2 Code to Convert Binary Date to ASCII

E. OTHER CONVERSIONS

The remaining conversions from ASCII to binary, or vice versa, follow the exact same pattern. The word tokens in the ASCII file are stored as their integer equivalents in the binary file. Integer numbers are stored as integers. Floating point values are stored as floating point values. ASCII character strings are stored as string size*char groupings, always preceded by an integer number indicating string size.

When converting from binary to ASCII all identifiers are automatically enclosed in double quotes, regardless of whether or not they are more than one word in length. Identifiers are titles, material names, light names or color names. They are stored as character strings in the binary format without the enclosing double quotes to save

space. Comments and date strings are written from binary to ASCII without double quote enclosure.

VI. THE PREVIEW PROGRAM

Preview is the name of the interactive 3D object viewing and modification program. **Preview** is located in the *off/tools/preview* directory. It was designed to permit manipulation of any object in an OFF ASCII or binary file. In this program, each object is read into the dynamic structure in order of decreasing element size, largest first, smallest last. Resolution levels, axis orientation, object size and polygon normals can all be interactively modified by use of this program. Additionally, the 3D object can be viewed in a "wire frame" adaptation of the normal solid object view. Appendix D contains a **preview** user's manual.

A. THE NEED FOR SUCH A TOOL

Standardization is inherently preferable to each programmer doing things in an individual manner. The OFF was developed so that 3D objects could be shared among simulations, present and future. However, an object used in one simulation might need to be scaled or otherwise modified for use in another simulation. For instance, one simulation might make vehicle or vessel course calculations based on all vehicles being oriented "facing north," or compass 0 degrees, in their file specification. Another might wish to base such calculations on geometric/mathematic 0 degrees, with vehicles "facing east" in their file specification.

Another common problem in the development of 3D objects for use with the IRIS's lighting capabilities is the calculation of polygon normals. There are many techniques used to calculate such normals. These methods rely on the specification of some "internal" or reference point in order to determine which of the two possible orientations a calculated normal should take. When an inside point is specified for a 3D

object, the vast majority of polygon normals are oriented correctly. However, some of the polygons may be oriented in such a way relative to the reference point that their normals are 180 degrees out of phase. These polygons with their reversed normals are easy to spot when the lighted object is viewed, as the polygon is black instead of its intended color. A way was needed to interactively identify these polygons and reverse their normals, that is, to re-orient the normal 180 degrees into the proper direction.

Finally, variable resolution display is a capability of **preview**. Interactively determining which polygons, lines and surfaces to show in a given object resolution is a highly useful feature. This feature allows the user to graphically "create" objects of lower resolution based on what is already in the object file. Permitting the user to modify the object as it is viewed in order to create varying object resolutions is an intelligent and meaningful way to create files of objects for different desired resolution levels in any simulation.

B. HOW OBJECTS ARE DISPLAYED IN PREVIEW

As an object is read in from the OFF, the dynamic structure discussed in chapter 4 is created. However, this time each line, polygon and surface is sized and put in the *draw list* in descending size order. Materials and colors are inserted in the *draw list* on an as needed basis, as explained in the following paragraphs.

The question arose as to how to determine line, polygon and surface sizes. The method being utilized at present is a rough approximation. First, all lines are considered to be of size zero (0), so that they are stored last in the *draw list* and displayed only at highest resolutions. Second, polygon and surface sizes are determined by using a rough approximation of their *volume,* determined by their respective minimum and maximum x, y and z vertex coordinates. As each polygon or surface is read from

the object data file, its vertices are sent to a calculation routine. This routine determines the minimum and maximum x, y and z values from all the vertices in that polygon/surface. The difference between the minimum and maximum values in all three directions is calculated. These differences are then squared, added together, and the square root of the sum taken. This final square root value becomes the polygon/surface size.

Resolution, in the sense of 3D objects, works much the same as the resolution levels utilized in the terrain display of current NPS simulators. Objects at a "distance" in the picture displayed do not need to be drawn in as great a detail. This is most readily accomplished by reducing the number of polygons, surfaces and lines drawn when displaying the object. It makes intuitive sense that the "smaller" polygons and surfaces would "disappear" in the distance first, and thus the "farther away" the object is in the display, the fewer "small" polygons or surfaces should be drawn. Lines should only be drawn when the highest resolution is desired.

The insertion of material or color nodes in the *draw list* is determined by the last such node encountered in the list and the most recent setting as ordered by the object data file. That is, a *current material* and a *current color* pointer are maintained by the program, and modified as **setmaterial** and **setcolor** tokens are encountered. When line, polygon or surface nodes are to be inserted in the *draw list*, the *draw list* is searched from the head forward, looking for the insertion position by element size. The routines used in this operation are found in files **insertelements.c** and **extrainsert.c**. As material and color nodes already in the list are encountered, a *list material* and *list color* pointer are maintained, showing the most recently encountered material or color in the *draw list.* When the insertion point for the line or polygon or surface is located, the *current material* (for polygons and surfaces) or *current color* (for lines) pointer is checked against the value of the *list* (color or material) pointer.

If they agree, the new node is simply added at that point. Figure 6.1 illustrates such an insertion.

In the event that the *current* and *list* pointers do not agree, two nodes must be inserted in the *draw list* around the new line/polygon/surface node. First, the *current material* or color must be added at the insertion point in the *draw list*, so the new line/polygon/surface is displayed in the appropriate color. Then, since the elements immediately following the new node are to be drawn in the present *list* material or color, a copy of the *list* node must be added after the new line/polygon/surface node. This completes the insertion, as Figure 6.2 illustrates.

A count of lines, polygons and surfaces is taken. These totals are displayed in the **preview** program help window. The total of polygons and surfaces is taken to determine the maximum *resolution_level* needed. This total is divided by the *resolution_factor*, an integer variable currently set to 10. Thus, for an object consisting of 127 polygons and 15 surfaces for a total of 142 elements, a maximum *resolution_level* of 15 (142/10, rounded up to include all elements) is required.

In the **preview** program, each line, polygon and surface structure has a field labeled 'active.' It is an integer field, used as a boolean value. Initially, all elements are marked as 'active' by asserting TRUE in all active fields. Thereafter, the mouse is used to increase or decrease resolution levels, or the interactive resolution menu and slider bars are used to select or delete elements from view. Lines, polygons and surfaces are marked as inactive (or reactivated) as chosen by the user. At all times, only elements whose 'active' field is TRUE are displayed in the viewport window.

The lighting model and lights used in the **preview** program are determined by a call to a set up routine. This routine is found in file **setup.c**. Default values are available and used for either the lighting model or the light(s) if either attribute is not specified in the object file.

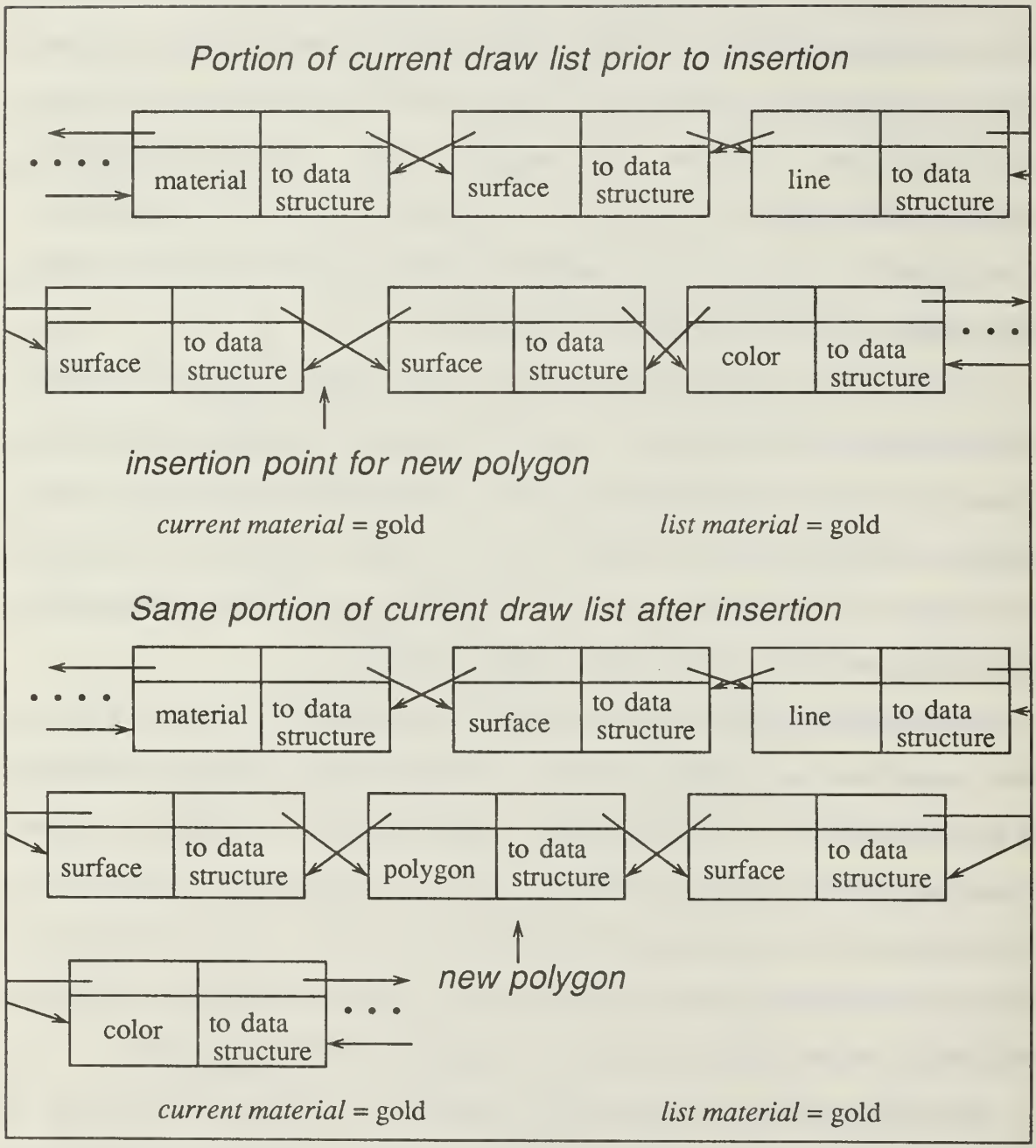

Figure 6.1 Node Insertion with Same Material Illustrated

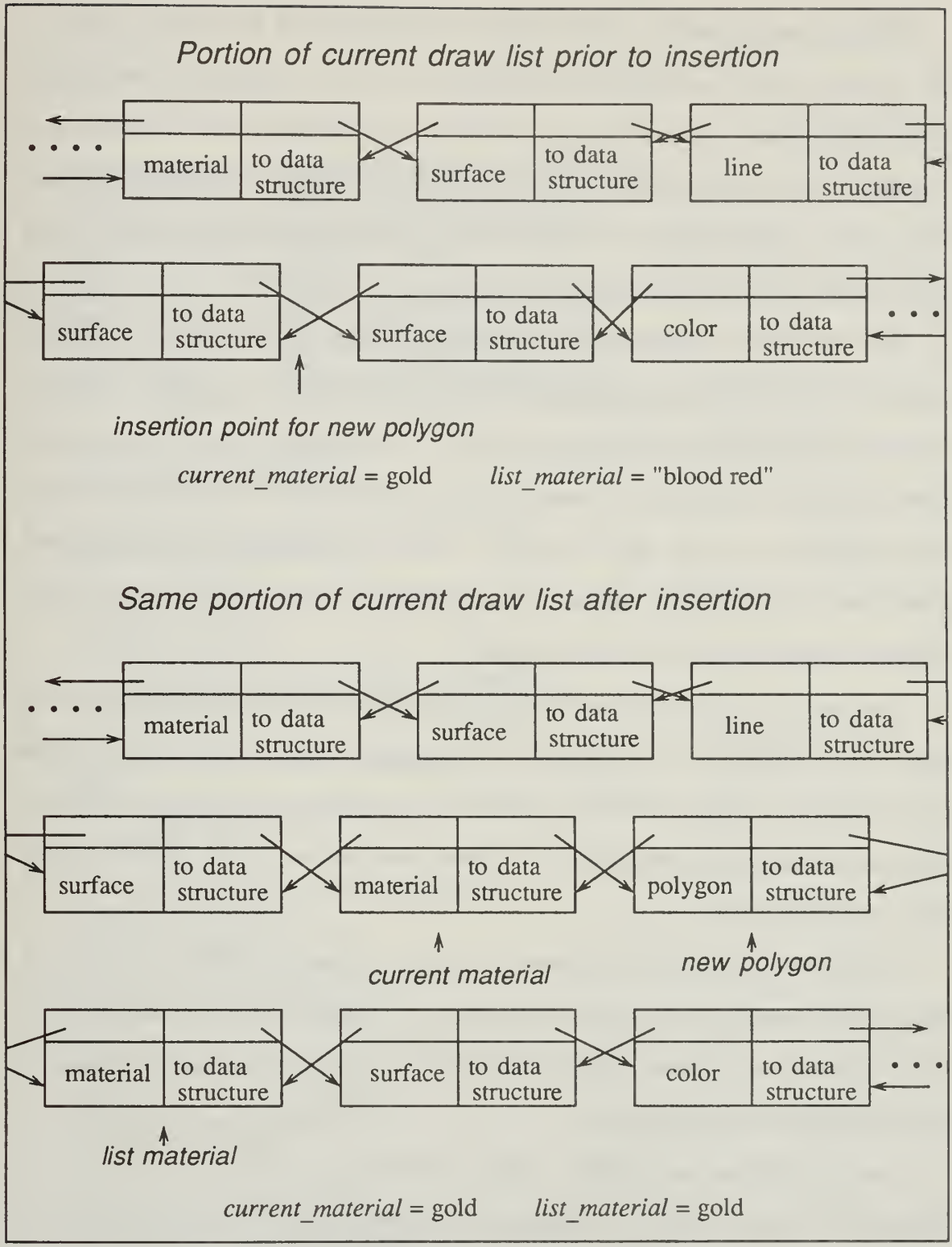

Figure 6.2 Node Insertion with Different Materials Illustrated

The first item operated on by setup() is the selection of the lighting model to use. If one or more is specified in the object file, the head pointer of the modeldefs list in the object's *header* structure will not be NULL. The lighting model (first model, if more than one are specified) is pointed to by the *head_modeldefs* list pointer. If none are specified, a default model is created for use. The current lighting model is then bound through IRIS lmdef() and lmbind() calls. Figure 6.3 presents the relevant code.

Next, the setup() routine finds and activates all lights specified in the object file. Currently, the IRIS lighting capabilities allow only eight lights to be active, so the set-up routine only activates the first eight lights in the *lightdefs* list, even if more are found in the list. If no lights are found in the list, a default light is created and inserted in the *lightdefs* list. Then, lights LIGHT0 .... LIGHT7 (maximum) are defined and bound based on the lights in the list, using IRIS lmdef() and lmbind() calls. Figures 6.4 and 6.5 show the code used for this operation.

The *current material* is defined and bound based on the first material found in the object's *materialdefs* list. As expected, if no materials were found in the file (hence none in the list), a default material is created for use. The *current material* is then defined and bound through use of the IRIS lmdef() and lmbind() calls. Figure 6.6 contains the relevant code.

Lastly, the *current color* is defined. If no colors were provided in the file, a default color is created and set as the current color. If one or more colors are defined in the object file, the first color in the *colordefs* list is set as the *current color*. Figure 6.7 contains the code for this operation.

Once the lighting model, lights, current material and current color have been established, the object is ready for displaying. The *draw list* is traversed in order from head to tail. Each node's *dtype* is checked and action taken appropriately.

```
int setup(hdr)

OBJECT_HEADER *hdr;

{
  float thismodel[10];   /* floating array of lighting model attributes */
  float thislight[14];   /* floating array of light attributes */
  float thismaterial[19]; /* floating array of material attributes */
  int theselights[9];    /* array of active lights */
  int i;                 /* loop counter */
  model_ptr temp;        /* pointer to model definition structure */
  light_ptr temp1;       /* pointer to light definition structure */
  material_ptr temp2;    /* pointer to material definition structure */
  color_ptr temp3;       /* pointer to color definition structure */
  double_ptr thisnode;   /* pointer to generic doubly-linked list node */

  /* set focus to viewport window */
  winset(windowlist[VIEWPORT]);

  /* set thisnode to first model node -- the one to use */
  thisnode = hdr->head_modeldefs;

  /* make sure there is a model definition provided */
  if (thisnode == NULL) {
     temp = initialize_model_def(temp);
     printf("No lighting model specified in Object File.\n");
     printf("Default lighting model created and used.\n");

  /* otherwise, assign to thisnode's data pointer */
  } else {
    temp = (model_ptr)DATA(thisnode),
  }  /* end if then else */

  /* now set attributes into model attribute array */
  thismodel[0] = AMBIENT;
  thismodel[1] = temp->ambient[0];
  thismodel[2] = temp->ambient[1];
  thismodel[3] = temp->ambient[2];
  thismodel[4] = LOCALVIEWER;
  thismodel[5] = temp->localviewer;
  thismodel[6] = ATTENUATION;
  thismodel[7] = temp->attenuation[0];
  thismodel[8] = temp->attenuation[1];
  thismodel[9] = LMNULL;

  /* set and bind current light model */
  lmdef(DEFLMODEL,CURRENTLMODEL,10,thismodel);
  lmbind(LMODEL,CURRENTLMODEL);
```

Figure 6.3 Lighting Model Setup Code

```
/* now to lighting */

/* set all lights in the array FALSE, inactive, initially */
for(i=0;i<10;i++)
  theselights[i] = FALSE;

/* assume all lights provided for in the object's file are to be */
/* active.  Therefore, bind the first eight definitions as active */

/* initialize counter */
/* start at 1 -- light index 0 defined by IRIS -- can't change */
i = 1;

/* if no lights in list, create default */
if (hdr->head_lightdefs == NULL) {
    temp1 = initialize_light_def(temp1);
    printf("No lights specified in Object File.\n");
    printf("Default light created and used.\n");
    double_insert(&hdr->head_lightdefs,(generic_ptr)temp1,LIGHTDEF);
}  /* end if */

/* set thisnode to first node in light def list */
thisnode = hdr->head_lightdefs;

while (thisnode != NULL) {

   /* set temp1 to current node's data pointer */
   temp1 = (light_ptr)DATA(thisnode);

   /* show light i is defined -- to be made active */
   theselights[i] = TRUE;

   /* set attributes of the light def array */
   thislight[0] = AMBIENT;
   thislight[1] = temp1->ambient[0];
   thislight[2] = temp1->ambient[1];
   thislight[3] = temp1->ambient[2];
   thislight[4] = LCOLOR;
   thislight[5] = temp1->lcolor[0];
   thislight[6] = temp1->lcolor[1];
   thislight[7] = temp1->lcolor[2];
   thislight[8] = POSITION;
   thislight[9] = temp1->position[0];
   thislight[10] = temp1->position[1];
   thislight[11] = temp1->position[2];
   thislight[12] = temp1->position[3];
   thislight[13] = LMNULL;
   lmdef(DEFLIGHT,i,14,thislight);
```

Figure 6.4 Partial Listing of Light Setup Code

```
  /* increment counter and move to next light in list */
   i++;
   thisnode = NEXT(thisnode);

}   /* end while */

/* for each light defined, do the binding to turn them on */
if (theselights[1])
   lmbind(LIGHT0,1);
if (theselights[2])
   lmbind(LIGHT1,2);
if (theselights[3])
   lmbind(LIGHT2,3);
if (theselights[4])
   lmbind(LIGHT3,4);
if (theselights[5])
   lmbind(LIGHT4,5);
if (theselights[6])
   lmbind(LIGHT5,6);
if (theselights[7])
   lmbind(LIGHT6,7);
if (theselights[8])
   lmbind(LIGHT7,8);
```

Figure 6.5 Remainder of Light Setup Code

```
/* now, set up first material definitions and bindings */

/* set thisnode to first material definition */
thisnode = hdr->head_materialdefs;

/* if no materials in list, get default */
if (thisnode == NULL) {
  temp2 = initialize_material_def(temp2);
  printf("No materials specified in Object File.\n");
  printf("Default Material Defined and Used throughout.\n");

/* otherwise, set temp2 to thisnode's data pointer */
} else {
  temp2 = (material_ptr)DATA(thisnode);
} /* end if then else */

/* now set material definition array characteristics */
thismaterial[0] = EMISSION;
thismaterial[1] = temp2->emission[0];
thismaterial[2] = temp2->emission[1];
thismaterial[3] = temp2->emission[2];
thismaterial[4] = AMBIENT;
thismaterial[5] = temp2->ambient[0];
thismaterial[6] = temp2->ambient[1];
thismaterial[7] = temp2->ambient[2];
thismaterial[8] = DIFFUSE;
thismaterial[9] = temp2->diffuse[0];
thismaterial[10] = temp2->diffuse[1];
thismaterial[11] = temp2->diffuse[2];
thismaterial[12] = SPECULAR;;
thismaterial[13] = temp2->specular[0];
thismaterial[14] = temp2->specular[1];
thismaterial[15] = temp2->specular[2];
thismaterial[16] = SHININESS;
thismaterial[17] = temp2->shininess;
thismaterial[18] = LMNULL;

/* define and bind current material */
lmdef(DEFMATERIAL,CURRENTMATERIAL,19,thismaterial);
lmbind(MATERIAL,CURRENTMATERIAL);

/* set global current material pointer */
current_material = temp2;
```

Figure 6.6 Current Material Setup Code

```
/* finally, set current color to first color, or default */

thisnode = hdr->head_colordefs;

/* if no colors in list, get default */
if (thisnode == NULL) {
   temp3 = initialize_color_def(temp3);
   printf("No colors specified in Object File.\n");
   printf("Default Color Defined and Used throughout.\n");

/* otherwise, set temp3 to thisnode's data pointer */
} else {
   temp3 = (color_ptr)DATA(thisnode);
} /* end if then else */

/* set global current color pointer */
current_color = temp3;
```

Figure 6.7 Color Setup Code

Nodes that are materials or colors cause the *current_materia*l or *current_color* global pointers to be reset to point to the new material or color. Polygons and surfaces, if active, are drawn using either standard IRIS 'bgnpolygon() ... endpolygon()' or 'bgnclosedline() ... endclosedline()' sequences. Whether polygons or lines are drawn is dependent upon whether or not the user has selected the wire frame option.

Prior to drawing any polygon or surface the *current_material* is checked against the *last_bound_material*. If the *current_material* is not the most recently bound material, a call to bind_current_material() is made to update the material definition being used in the IRIS graphics pipeline.

Lines can not be drawn with lighting active on the IRIS, so when lines are encountered lighting is turned off. Lines are then drawn in the *current_color* by issuing a c3f() call to the IRIS. Each line segment specified in the line is drawn by the standard 'bgnline() ... endline()' sequence. Lighting is then again turned on by a call to lmbind() with the CURRENTLMODEL. When polygons and surfaces are drawn as lines due to the wire frame option, the color provided to the c3f() call is the diffuse color specified for the *current_material*. Lighting is turned off when the wire frame option is selected and turned back on when it is deselected.

C. THE OBJECT ORIGIN

All objects are assumed to be centered about the origin, (0.0 0.0 0.0), unless otherwise specified by an **origin** token in the object file. The object is displayed in the viewport window such that the object origin is at the center of the graphics window, which in this case is the 3D origin, (0.0 0.0 0.0). When the object is drawn, a pushmatrix() and translate() are ordered to position the object's origin at this center. The translate takes as its arguments the negative (reverse) values of the specified object

origin. After the object is drawn, a popmatrix() call returns the stack to its pre-drawing status.

D. MODIFYING ACTUAL OBJECT DATA

The actual object data can be manipulated in the three basic ways -- translation, rotation and scaling. Rotation and translation can be done about or along any single axis at one time, and scaling can be done along any single axis or along all three axes simultaneously.

Since the actual modification of the object data is done for the purpose of "creating" a new object, these operations require great precision. As this is the case, slider bars are not used for input in data modification. Instead, in each instance a new window appears, seeking keyboard input of exact values to use in modifying the data. For rotation, the input is expected in floating point degrees, either positive or negative. Rotation follows the "right hand rule" for determining positive direction about each axis. For translation, the input is expected to be a floating point number. For scaling, the input is expected to be a floating point number other than 0.

Once the input is received, the *draw list* is traversed, and scaling, translation or rotation is performed on each vertex of each element (line, polygon, surface). For rotation, the normal for each polygon and each surface vertex is also modified. If negative scaling is done, the polygon and vertex normals are reversed. For translation, the object's origin is also modified. The object is then re-displayed in the viewport window. The user will note the updated minimum and maximum values displayed in the help window, and, in the case of translation, the updated object origin.

E. RESOLUTION LEVELS

The activation of elements and the overall resolution scheme was presented in section B of this chapter. This section discusses the actual method of modifying resolution levels.

When the main help and tools windows are visible, the help window display indicates that the left mouse decreases the resolution level and the middle mouse increases the resolution level. Each decrement in the resolution level removes one *resolution_factor*'s worth of elements from the display, always removing the smallest elements first. Each increment in the resolution level brings back one *resolution_factor*'s worth of elements to the display. The current *resolution_level* is displayed at the bottom of the main help window.

Once these mouse buttons have given the user a rough approximation of the resolution desired, the actual resolution can be fine tuned by use of the RESOLUTION option in the main tools window menu. Selection of this option puts two slider bars in the tools window, and the appropriate display in the help window. The left slider is for selecting polygons; the right slider for selecting surfaces.

Only the most recently used slider bar will have its current element highlighted. That is, if the polygon slider bar is moved, the current polygon is highlighted. If the surface slider bar is moved, the current surface is highlighted. The current polygon is always highlighted in white; the current surface is always highlighted in yellow. Each is also ringed by a red border line drawn around its perimeter. Use of the middle mouse increments BOTH sliders simultaneously, one element for each press of the button.

All polygons and surfaces in the object are highlighted as they are selected, even those not currently ACTIVE in the display. To see an element not currently

displayed, highlight it and choose SELECT THIS (polygon/surface) from the menu now available in the tools window. Selecting an element makes it active until it is deleted by another menu selection or by reducing resolution again at the main level. To delete an element from the picture, highlight it and choose DELETE THIS (polygon/surface) from the menu available in the tools window. Deleting an element makes it inactive until it is SELECTED to be active again by the menu, or by an increase in resolution level performed at the main level. Continue in this fashion until only the desired elements are actively displayed in the viewport window. The user can exit the resolution option to rotate or translate the object for a different view, then reenter it to continue modifying the resolution.

F. SAVING THE MODIFIED OBJECT

There are two options for saving the previewed object to a new file, both available at any time from the menu in the viewport window. These options both write the output to a file with the same name as the input object file but with the tag ".new" appended to it. The output file has the same format, ASCII or binary, as the input file.

The first option saves ALL the current object data, even those elements not currently displayed. This is the option to select if the entire object has been rotated, scaled or translated.

The second option saves only those elements currently displayed in the viewport window. This option is used for saving an object in a different resolution, regardless of whether it has been otherwise transformed or not.

It is important to note that this saving option can be invoked repeatedly throughout the course of the **preview** program. If a mistake is made in saving a file that needs to be further modified, the modifications can be made and the object saved again. The output file always contains the most recently saved version. Hence, to

make multiple resolution copies from one use of **preview**, the output file should be moved to a different name by use of another terminal prior to performing any additional saves.

VII. INTEGRATING OBJECTS INTO OTHER PROGRAMS

The primary goal of this research was the development of a file format for 3D objects that would permit the display of objects in any simulator or other program. The **preview** tool and its associated routines contain some level of complexity beyond that simple goal, in order to permit object modification. This chapter presents the simpler routines needed to integrate objects in the OFF format into other programs.

A. THE MAJOR DIFFERENCES

The major differences between the routines used in the **preview** tool and normal program integration lie in the fields present in various element structures. The structures used in the routines needed for integrating objects into another program are simpler than those structures utilized by the **preview** tool. Also, the object does not have its lines, polygons or surfaces sized and inserted in the *draw list* by size order.

The line, polygon and surface structures, defined in file **filespec.h**, each lack the 'size' and 'active' fields found in the **preview** tool. The **filespec.h** file found in the *off/3dintegration* directory has these structures redefined from those in the same file in the *off/tools/preview* directory. Similarly, the routines found in the file **initializers.c** in the two directories reflect the different structures defined. The initializer routines create new structures when called, and assert default values into each field of the structure. Figures 7.1 and 7.2 contain excerpts of the relevant code from the files in the *off/3dintegration* directory.

The main object drawing routine, drawobject(), is found in the file **drawobject.c**. In the integrated version, there is no need to check the active field of lines, polygons or surfaces. In fact, such fields do not exist. Nor is there any option to draw the

```
struct polygon_def   /* polygon of vertices and a normal */
{
   int               numitems;  /* number of vertices plus normal */
   float             normal[3]; /* polygon surface normal xyz */
   two_dim_array     data;  /* pointer to start of data triples */
};

struct line_def   /* line defined by vertices to call to bgnline/endline */
{
   int               numitems;  /* number of vertices plus normal */
   two_dim_array     data;      /* pointer to start of data triples */
};

struct surface_def   /* list of surface definitions */
{
   int               numitems;  /* number of data sextuplets */
   two_dim_array     data;      /* pointer to data list */
};
```

Figure 7.1 Polygon, Line and Surface Structure Definitions

```
/* this routine allocates a surface def structure, puts default values in
   its slots, and returns a pointer to it as its value.              */
surface_ptr initialize_surface_def(ptr)

surface_ptr ptr;

{
  ptr = (struct surface_def *)malloc(sizeof(struct surface_def));
  ptr->data = NULL;
  ptr->numitems = 0;
  return(ptr);
}  /* end initialize_surface_def */

/* this routine allocates a polygon def structure, puts default values in
   its slots, and returns a pointer to it as its value.              */
polygon_ptr initialize_polygon_def(ptr)

polygon_ptr ptr;

{
  ptr = (struct polygon_def *)malloc(sizeof(struct polygon_def));
  ptr->data = NULL;
  ptr->numitems = 0;
  ptr->normal[0] = 0.0;
  ptr->normal[1] = 0.0;
  ptr->normal[2] = 0.0;
  return(ptr);
}  /* end initialize_polygon_def */

/* this routine allocates a line def structure, puts default values in
   its slots, and returns a pointer to it as its value.              */
line_ptr initialize_line_def(ptr)

line_ptr ptr;

{
  ptr = (struct line_def *)malloc(sizeof(struct line_def));
  ptr->data = NULL;
  ptr->numitems = 0;
  return(ptr);
}  /* end initialize_line_def */
```

Figure 7.2 Initializer Code for Lines, Polygons and Surfaces

object in a wire frame version. Hence, these checks have been removed from the **drawobject.c** version found in the *preview* directory. In the integrated version, all elements of the object are drawn each time drawobject() is invoked.

B. THE ROUTINES NEEDED

All of the routines that might be needed are included in the directory *off/3dintegration*. It is possible that not all files will be needed, as outlined in the following paragraphs.

Every program wishing to integrate 3D objects from OFF files will require the following routines: **filespec.h**, **filelex.c**, **allocatearray.c**, **drawobject.c**, **getfloats.c**, **initializers.c**, **insertelements.c**, **listroutines.c**. Additionally, at least one of the series of routines to read either an ASCII or a binary format file must be included. These are presented below. There are a small number of global variables defined in the file **filespec.h**. These must be checked to ensure they do not conflict with any other variables chosen for the user's program(s). Additionally, each of the included files should be checked for the names of routines they contain. A listing of all such routines is provided at the head of each file. Again, conflicts must be avoided between function names. Finally, for the global variables to be properly defined in the main program routine, the statement *#define MAIN* should be included in the C language module in which the user has defined the *main()* function. Figure 7.3 illustrates.

The routines needed to read an ASCII file are contained in the files **readascii.c** and **asciiread.c**. Similarly, the routines to read from a binary file are contained in **readbinary.c** and **binaryread.c**. If the user knows for certain that he will only be reading from one such file type, only those routines need be included. If the user is unsure, or may be reading from both types within his program, he should then include the file **checkfiletype.c**. This file contains routines that do all necessary checking for

file type, and invoke the appropriate reading routines based on that type. Figures 7.4 and 7.5 show the file type checking code.

```
 #define MAIN        /*define MAIN first for global variable recognition */
#include filespec.h  /* has definitions needed of variables and structures */

/* reminder to link together with my program the compiled versions of
   filelex.c  drawobject.c  allocatearray.c  getfloats.c
   initializers.c  insertelements.c  listroutines.c
  or they may be included here */

/* my main simulator routine */
main()
{
. . . . . (general program code)
}
```

Figure 7.3 Sample Beginning Program Code with Included Files

One final file may be of interest. The file **setup.c** contains code to bind the lighting model, lights, current material and current color used to those specified in the object file. More often than not, most simulations will already have their model and lights defined, and will only be interested in drawing objects under the established conditions. However, should the user wish to use the lighting model and light(s) defined in the file, they should include this routine as well. With this file included, a call to the routine setup() prior to drawing the object will achieve the desired result.

C. ACTUAL IMPLEMENTATION

As each object is read in from a standard object file, it is stored in a series of linked lists associated with an object header structure. Therefore, the process of reading and displaying objects is extremely straightforward.

One OBJECT_HEADER should be defined for each object desired for display in the program. Then, at some point prior to the actual graphics display loop in the

```
/* this routine checks a file to see if it is an ASCII or binary type */
int check_file_type(filename)

char *filename;
{
  char filestr[90];   /* string to be created from filename */
  char instrings[90];    /* place to read from temp file */
  FILE *infile;       /* FILE pointer to input temp file */
  int whichtype = UNKNOWNFILE; /* integer flag for file type */
  int notdone = TRUE;         /* flag for loop control */

  /* copy filename to filestr, append output to junkfile */
  strcpy(filestr,"file ");
  strcat(filestr,filename);
  strcat(filestr," > checktypejunkfile");

  /* invoke call to "file" at system level */
  system(filestr);

  /* now, go read what "file" did! */
  infile = fopen("checktypejunkfile","r");
```

Figure 7.4 First Section of File Type Checking Code

```
while (notdone) {

  /* scan file -- check for EOF by fscanf returning < 0 */
  if (fscanf(infile,"%s",instrings) < 0)
    notdone = FALSE;

  /* check for ASCII or DATA key words being read */
  if (!strcmp(instrings,"ascii")) {
    notdone = FALSE;
    whichtype = ASCIIFILE;
  } else if (!strcmp(instrings,"data")) {
    notdone = FALSE;
    whichtype = BINARYFILE;
  } /* end if then else */

 }  /* end while */

 /* close the file */
 fclose(infile);

 /* via system level call, remove temp junk file */
 system("rm checktypejunkfile");

 /* return result */
 return(whichtype);

}  /* end check file type */
```

Figure 7.5 Remainder of File Type Checking Code

program, invoke calls to the appropriate file reading routines. The choices include readascii(), readbinary(), or read_objectfile(). Each of these routines takes as their arguments a *hdr* pointer (where *hdr* is declared as: OBJECT_HEADER *hdr) and a file name *fname* (where *fname* is declared as: char *fname). Figure 7.6 shows the code for the routine read_objectfile().

Thereafter, setup() can be used to establish the IRIS lighting model and light(s) as defined in the object file of choice, or the object(s) can be drawn using lighting conditions otherwise established for the program. Graphics focus should be set to the desired display window, and all translations and rotations performed so that the system transformation stack has the object origin positioned in the desired location. At that point, a call to drawobject(), with a *hdr* pointer as argument (again, *hdr* is declared as: OBJECT_HEADER *hdr) will draw the specified object in the graphics window. This process can be repeated in different positions for multiple copies of a single object or in various positions with various objects to obtain the desired picture. Figure 7.7 provides some sample code.

One important note should be observed at this point. The origin of the object is accounted for by the drawobject() routine. That is, if the specified origin of the object in the file is other than (0.0 0.0 0.0), the drawobject() routine performs appropriate transformations so that the object is drawn about its specified origin at the location desired. In other words, the user should not be concerned with obtaining the object's origin and trying to account for it in the program prior to calling drawobject().

```
/* this routine lets the user specify an object file to read, and */
/* reads that file from memory into the object structure */
/* pointed to by hdr.  It initializes the object header, and calls a */
/* routine that determines the input file type. */
int read_objectfile(hdr,filename)

OBJECT_HEADER *hdr;
char *filename;

{
 int whichtype;   /* flag to indicate type of input file */

 /* initialize header */
 initialize_header(hdr);

 /* get file type for filename */
 whichtype = check_file_type(filename);

 /* now, read into hdr based on file type */
 switch (whichtype) {
  case ASCIIFILE:
    readascii(hdr,filename);
    return(TRUE);
    break;
  case BINARYFILE:
    readbinary(hdr,filename);
    return(TRUE);
    break;
  case UNKNOWNFILE:
    printf("\nMAJOR ERROR!!\n");
    printf("Can not read file >>> %s\n",filename);
    printf("\nFile must be in standard ASCII or BINARY file format.\n");
    printf("Check filename and try again.\n\n");
    return(FALSE);
    break;
 } /* end switch */

} /* end read objectfile */
```

Figure 7.6 Read Object File Code

```
/* sample use of integrated object drawing routines */

/* we have assumed that the necessary files have been linked at compile
   time of the main program */

#define MAIN
#include filespec.h

/* my simulator code */
main()
{

  OBJECT_HEADER mysubmarine;
  OBJECT_HEADER mycarrier;
. . . . . (other declarations)

. . . . . (preliminary simulator code prior to graphics display routines)

  /* I have an object in some format in file Submarine.ohioclass */
  read_object(&mysubmarine,"Submarine.ohioclass");

  /* I know Kittyhawk.ascii is in ASCII file format */
  readascii(&mycarrier,"Kittyhawk.ascii");

. . . . . (more setting up of the simulator)

   /* now, with the graphics window open and the proper translations */
  /* and rotations done for object positioning, I can display my sub */
  drawobject(&mysubmarine);
. . . . (other display code)

   /* Now I wish to display my carrier at the current graphics position */
  drawobject(&mycarrier);

. . . . . (rest of simulator code)

}
```

Figure 7.7 Sample Use of Integrated Routines and Object Display

VIII. LIMITATIONS OF THE FILE FORMAT AND TOOLS

The initial conception of what the standard file format should be and might accomplish was different than the finished product. Perhaps these limitations can be overcome by future research, or perhaps the expectations must be modified.

A. RESOLUTION

The very first conception of object resolution was that only one file needed to be created for each 3D object. Thereafter, the object would be drawn to the desired resolution by specifying some resolution level or factor when invoking the drawing routine.

The approach envisioned to accomplish this was the sizing utilized in the **preview** tool. That is, order elements in an object by size and draw only some specified number of the largest ones at various resolutions. However, it quickly became apparent that this would create visible holes in objects that would be obtrusive and inappropriate for viewing.

Two ways were considered for overcoming this shortcoming. The first was to somehow mark each element with the lowest resolution at which it should be shown. Any resolution specified below that level would not display this element. This created a great difficulty in determining how to specify such a marking, how to set the marking for each element when creating or editing a file, and resulted in more information having to be stored for each object.

The second alternative was the one decided on and implemented in the **preview** tool. If more than one resolution of a given object were desired, that resolution could be created interactively from a base object file using the **preview** program, then the new resolution could be saved. Thus, when an object file is read into another program

or simulation for display, the entire object is displayed at all times. This eliminates the need for any special marking of object elements, including the elimination of the 'active' and 'size' fields of all element structures. Regrettably, this means a different file must be created for each desired resolution level of a given object.

B. AUTOMATIC OBJECT GENERATION

None of the tools here lead directly to automatic generation of new 3D objects from "parts" in existing files. However, it is not hard to envision how many aspects of the **preview** selection and saving mechanisms could greatly aid such construction.

Assuming ASCII 3D object files are adequately commented, it would be relatively easy to copy an existing object ASCII file to a new file, then delete all but a select portion of that file. Hence, from a file containing an entire submarine, for example, a file containing only a conning tower, or main hull, or nose section, etc., could be created by simple use of a text editor.

Once enough objects were thus "dismantled" into piecemeal files, new objects could be created by liberal use of the ASCII format's **include** feature in a new object file. Further, each individual object "piece", in its own file, could be re-oriented, scaled, and/or translated to created additional "building block" files. It is understood that forthcoming research at NPS will look into such construction of objects from "building block" files. Clearly, the standard file format has laid the groundwork for such object construction.

C. PERFORMANCE DEGRADATION

Obviously, it takes longer to read an object from an ASCII or binary file to a dynamically created structure than to access an object coded directly into the program or simulation. However, the actual time to access an object file and store the data into

the dynamic structure is minimal compared to the time to read the current data bases from which the simulations draw their terrain data. Furthermore, all such terrain and object file accessing takes place during simulation initialization and need never be done again throughout the simulation run. There is no noticeable difference in performance in drawing the object from the dynamic structure as opposed to drawing it from a data structure within the simulator code.

IX. CONCLUSIONS AND FUTURE WORK

Perhaps the most surprising aspect of this research has been the discovery of the ease with which all vehicles, vessels and other 3D objects currently in use for NPS simulations can be converted to the standard file format. Currently, all platforms used in the CCWF Submarine and Periscope View simulator have been converted to the standard format, along with several other platforms from other NPS simulations and IRIS demo programs. These objects can be found in the *off/3dobjects* directory.

Developing material definitions, and trying them on various platforms in the **preview** tool program, has been extremely informative and beneficial. In a matter of moments, by simple additions of **defmaterial** and **setmaterial** elements in the object file, whole new materials can be tried for their suitability in general, and in the specific object in particular.

The **preview** tool has proved invaluable at finding and correcting errant polygon normals in 3D objects. In fact, the current CCWF Submarine and Periscope View simulation uses hard coded polygon normals that were verified and corrected by use of the **preview** tool. At present, no NPS simulation has integrated the use of standard object files into their programs.

The conversion of the ASCII file format to binary has achieved a significant file size compression. Typically, depending on the number and type of tokens in the ASCII file, a binary file size of between 45% and 48% of the ASCII file is achieved. No specific goals for compression were set at the inception of this research, but a reduction of better than 50% is considered significant.

As was mentioned, no current NPS simulation has integrated standard file format objects into their displays. This is mainly because each simulator is currently the subject of ongoing research in other areas, such as basic simulator development or networking. Future work in the OFF area should quickly concern itself with integration of objects into current and future NPS simulations.

Finally, future effort could well be placed on developing some interactive tool to construct new 3D objects from files of object "pieces", such as hulls, tracks, bodies, superstructures, etc. Along with this, the development of a library of lights, lighting models, material definitions, color definitions, and object "pieces" that have been tried and found useful should be constructed. Files in this library could easily be included in future object files.

APPENDIX A

THE ASCII FILE FORMAT

The ASCII Object File Format (OFF) is a text file format which is case distinct. The file consists of token strings and associated data values which are processed in order from file beginning to EOF in one pass. The program utilized to accomplish this is the lexical analyzer module **filelex.c**, created by use of the UNIX system Lex tool. Tokens are read by calling the routine *yylex()*. Figure A.1 illustrates a sample use of this function.

The file tokens are presented in the following paragraphs. Items in **boldface** are to be typed as is, *italics* denote user supplied data. Names and strings are sequences of up to 200 characters which must be enclosed in double quotes ("") if the sequence contains blanks/spaces. All numbers in parenthesis are the number of floating point values expected after the given property token, except where noted.

It is imperative that all floating point values contain a decimal point. Numbers without decimal points will be treated as integers, and will cause errors when floating point numbers are expected. For clarity, it is recommended, though *not required*, that floating point numbers have at least one digit in front of the decimal point, as in 0.3 instead of .3. Whole floating point numbers, such as 12, must be specified as 12.0.

Any group of floating point values expected together should be treated as a *block* of data and should not be separated by comments or any other tokens.

title *object_name* : *object_name* is a string giving the name of the object defined in the file.

```
/* this routine writes the ASCII file title to the binary file */
write_binary_title(bf)

int bf;

{
 int   nexttoken;  /* next token in the input stream */
 int   firsttoken = TOK_TITLE;
 int   titlesize;

 /* get next token from input file */
 nexttoken=yylex();

 /* should be an ID -- a "string" */
 if (nexttoken==TOK_ID) {
   titlesize = strlen(id);
   write(bf,&firsttoken,sizeof(int));
   write(bf,&titlesize,sizeof(int));
   write(bf,id,sizeof(char)*titlesize);

 /* if not, error! */
 } else {
   printf("ERROR -- no identifier after key token TITLE \n");
   printf("No title written to binary file\n");
 }
 return(1);
}  /* end write binary title */
```

Figure A.1 Sample Use of *yylex()* Routine

date *date_string* : string defining the creation date or last modification date of the file. The date string is of the form dd mmm.... yyyy, where dd is a integer date (day), mmm... is a text string for the month, and yyyy is any integer number indicating the year.

include *file_name* : shift reading to new file given by *file_name*. If file isn't found on current search path, then look in the specified default directory (specified within the tools programming). *File_name* must conform to UNIX file naming rules.

deflight *light_name light_properties* : specifies that the tokens and data up to the **defend** token define properties of the light. Any properties which are not provided in the file's light definition will be set to default values. Thus, not all properties need be specified. The property tokens are given below. They have the same meaning as that given in the IRIS manuals.

ambient (3) define the light's contribution to the ambient scene light.
lcolor (3) define the light's color.
position (4) define the light's position or direction.
defend (0) specifies the end of the current definition.

deflmodel *model_properties* : tokens up to **defend** define properties of a lighting model. Not all properties need be specified, as in light definitions above. The following properties are allowed:

ambient (3) ambient light in the model.
localviewer (1) whether viewer position is local or infinite.
attenuation (2) lighting model distance attenuation factors.
defend (0) end of current definition.

defmaterial *material_name material_properties* : tokens up to the next **defend** define the properties of the material to be associated with *material_name*. The following tokens are the valid material properties. Once again, not all need be specified.

emission (3) define the material's emission color
ambient (3) define the material's ambient contribution color
diffuse (3) define the material's diffuse component color
specular (3) define the material's specular highlighting component color
shininess (1) define the material's shininess factor
alpha (1) define the material's alpha value
defend (0) end of current definition.

defcolor *color_name r g b* : defines a color for use in drawing object lines. The *r*, *g* and *b* values are floating pointer numbers between 0.0 and 1.0, and will be utilized in an IRIS c3f() call.

setmaterial *material_name* : used to set the material for all following polygons and surfaces until changed by another **setmaterial**.

setcolor *color_name* : used to set the color for all following lines until another **setcolor** is issued.

defline # *vertex_list* : used to specify a sequence of line segment vertices that will be joined by the Iris command series 'bgnline ..... endline'. Each vertex will be joined to its predecessor by a line segment. For closed lines, repeat the first vertex at the end of the line definition. The # is an integer value depicting the number of vertices, and the *vertex_list* is a list of x y z triples specifying the x y z coordinates of each vertex.

defpoly *normal* # *vertex_list* : used to specify a polygon defined by three or more vertices and having a single surface normal. Here *normal* is the x y z components of the polygon's normalized (unit length) normal vector, # is the number of vertices defining the polygon, and *vertex list* a list of x y z triples specifying the x y z coordinates of the polygon vertices. Note that # does not include the *normal* in its count.

defsurface # *vertex_list* : similar to **defpoly,** but it defines a surface with vertex normals, where # is the number of vertices and *vertex list* consists of a series of x, y, z, i, j, k sextuples defining the x, y, z coordinates and the i, j, k (unit length) normal components.

origin *x y z* : defines a new origin about which the object is centered. That is, the default object center is (0.0,0.0,0.0), but the object may be centered elsewhere by including this option. This does not do a translate but merely relates the coordinates in the file to the actual origin.

/* *text* */ : denotes comments as in C. Nesting of comments is not allowed. Comments are allowed on separate lines, or at the end of a line, or anywhere comments would normally be allowed in C. Comments within a *block* of floating point values will be ignored and lost when converting an ASCII file to binary format. That is, any time *n* floating point values are expected, such as after tokens like **lcolor, ambient, emission,** etc., or after **defcolor** *name*, or **defline** #, there should not be any comments in those areas. Such comments will be lost when converting from ASCII to binary formats and back. In short, do not put comments *inside* of **defpoly, defline, defcolor** or **defsurface** declarations. Comments before or after these are perfectly valid. Comments within **defmaterial, deflmodel,** or **deflight** should be placed before or after keywords or at ends of lines, but not between expected float values. Also, the *yylex()* code will not recognize any comment that is more than 200 characters long. Therefore it is suggested that all comments be limited to at most two lines.

Figures A.2 and A.3 contain a sample file in the ASCII OFF format. This object file, though meaningless if displayed, shows each of the object attributes in use.

```
title  "A Sample File Full of Garbage"

date 15 May 1989

/* define a material and a color */
defmaterial   subsilver
ambient 0.400000 0.400000 0.400000
diffuse 0.300000 0.300000 0.300000
specular 0.900000 0.900000 0.950000
shininess 30.000000
defend

defcolor  mytrialcolor
0.8  1.0  0.4

/* specify object origin */
origin 0.0 0.0 0.0

/* sample use of another defined file, containing more materials */
include  mymaterials.sample

/* set color and material */
setcolor mytrialcolor

setmaterial subsilver

/* sample light */
deflight redlight
ambient 0.1 0.5 0.5
lcolor 0.0 1.0 0.0
position 13.0 35.0 10.0 1.0
defend
```

Figure A.2 Partial Sample 3D Object File in ASCII OFF Format

```
/* sample lighting moded defintion */
deflmodel
ambient 0.2 0.2 0.2
localviewer 1.0
attenuation 1.0 0.5
defend

/* define a surface */
defsurface
4
-20.000000 20.000000 20.000000 -0.333333 0.333333 0.333333
-20.000000 20.000000 -20.000000 -0.333333 0.333333 -0.333333
-20.000000 -20.000000 -20.000000 -0.333333 -0.333333 -0.333333
-20.000000 -20.000000 20.000000 -0.333333 -0.333333 0.333333

/* define a polygon */
defpoly
-0.402739 0.000000 0.915315
3
0.000000 0.000000 0.000000
-0.500000 0.110000 -0.220000
-0.500000 -0.110000 -0.220000

/* define a line */
defline
3
4.593  2.2345 1.11197
0.0  0.0  0.0
1.234 4.567  3.34283
```

Figure A.3 Completion of Sample 3D Object File in ASCII OFF Format

APPENDIX B

THE BINARY FILE FORMAT

The OFF binary format relies upon associating integer values with tokens of the ASCII format. These integer tokens are defined in the file **filespec.h** and are repeated below. Each integer token has associated with it a predetermined amount of data in a specified format, as outlined in the following paragraphs.

Strings stored in the binary format are not enclosed in double quotes, even if they contain blanks/spaces. This is due to the fact that the binary file reading routines always take as input a specified number of characters and do not rely on the quotes to identify the beginning and ending of strings. Furthermore, the file conversion tool **fileconvert** automatically encloses all identifier strings (titles, material names, color names, light names) in double quotes when performing binary to ASCII conversion.

A. THE BINARY TOKENS

The following token definitions have been established in file **filespec.h** and are thus provided to all routines concerned with reading or manipulating data or files in the standard object format.

```
#define TOK_TITLE    1
#define TOK_DATE     2
#define TOK_COMMENT  3
#define TOK_DEFLIGHT  20
#define TOK_DEFMATERIAL  21
#define TOK_DEFLMODEL  22
#define TOK_DEFCOLOR  23
#define TOK_DEFPOLY  24
```

```
#define TOK_DEFSURFACE   25
#define TOK_DEFLINE   26
#define TOK_ALPHA   30
#define TOK_AMBIENT   31
#define TOK_ATTENUATION   32
#define TOK_DIFFUSE   33
#define TOK_EMISSION   34
#define TOK_LCOLOR   35
#define TOK_LOCALVIEWER   36
#define TOK_POSITION   37
#define TOK_SHININESS   38
#define TOK_SPECULAR   39
#define TOK_DEFEND   40
#define TOK_SETMATERIAL   50
#define TOK_SETCOLOR   51
#define TOK_ORIGIN   60
```

B. LAYOUT OF THE BINARY FILE

The allowable items in the binary file have no specified order. That is, any item may be placed in the file anywhere and it will not effect the reading of the file. However, polygon, surface or line colors are dependent upon where they appear following the most recent **set-material** or **setcolor** tokens. Consult Appendix A for further clarification.

The allowable items, and their required formats, are outlined below. The file reading routines expect any binary file to have items in these formats, and deviations will cause errors. Unless otherwise noted, order of elements within a specified item is important. Order is not entirely important in lightdefs, modeldefs and materialdefs, as outlined in the following.

Title:

TOK_TITLE	sizeof(int)	
Title Size	sizeof(int)	# characters in title
Title String	sizeof(char)*Title Size	character string title

Date:

TOK_DATE	sizeof(int)	
Date Size	sizeof(int)	# characters in date
Date String	sizeof(char)*Date Size	character string date

Comment (/* .... */):

TOK_COMMENT	sizeof(int)	
Comment Size	sizeof(int)	# characters in comment
Comment String	sizeof(char)*Comment Size	character string comment

Light Definition:

TOK_DEFLIGHT	sizeof(int)	
Name Size	sizeof(int)	# characters in light's name
Light Name	sizeof(char)*Name Size	light name string

-- the following blocks may occur in any order until TOK_DEFEND --
-- comment block may occur repeatedly until TOK_DEFEND --

TOK_COMMENT	sizeof(int)	
Comment Size	sizeof(int)	# characters in comment
Comment String	sizeof(char)*Comment Size	character string comment
TOK_LCOLOR	sizeof(int)	
Color Array	sizeof(float)*3	
TOK_AMBIENT	sizeof(int)	
Ambient Array	sizeof(float)*3	
TOK_POSITION	sizeof(int)	
Position Array	sizeof(int)*4	
TOK_DEFEND	sizeof(int)	

Model Definition:

TOK_DEFLMODEL	sizeof(int)

-- the following blocks may occur in any order until TOK_DEFEND --
-- comment block may occur repeatedly until TOK_DEFEND --

TOK_COMMENT	sizeof(int)	
Comment Size	sizeof(int)	# characters in comment
Comment String	sizeof(char)*Comment Size	character string comment

TOK_LOCALVIEWER	sizeof(int)
Localviewer Value	sizeof(float)

TOK_AMBIENT	sizeof(int)
Model Ambient Array	sizeof(float)*3

TOK_ATTENUATION	sizeof(int)
Attenuation Array	sizeof(float)*2

TOK_DEFEND	sizeof(int)

Material Definition:

TOK_DEFMATERIAL	sizeof(int)	
Name Size	sizeof(int)	# characters in Material Name
Material Name	sizeof(char)*Name Size	Material Name String

-- the following blocks may occur in any order until TOK_DEFEND --
-- comment block may occur repeatedly until TOK_DEFEND --

TOK_COMMENT	sizeof(int)	
Comment Size	sizeof(int)	# characters in comment
Comment String	sizeof(char)*Comment Size	character string comment

TOK_EMISSION	sizeof(int)
Emission Array	sizeof(float)*3

TOK_AMBIENT	sizeof(int)
Ambient Array	sizeof(float)*3

TOK_DIFFUSE	sizeof(int)
Diffuse Array	sizeof(float)*3

TOK_SPECULAR	sizeof(int)

Specular Array	sizeof(float)*3	
TOK_SHININESS	sizeof(int)	
Shininess Value	sizeof(float)	
TOK_ALPHA	sizeof(int)	
Alpha Value	sizeof(float)	
TOK_DEFEND	sizeof(int)	

Color Definition:

TOK_DEFCOLOR	sizeof(int)	
Name Size	sizeof(int)	# characters in Color Name
Color Name	sizeof(char)*Name Size	Color Name string
Color Array	sizeof(float)*3	Color values 0-1

Origin:

TOK_ORIGIN	sizeof(int)	
Origin Array	sizeof(float)*3	

Setcolor Definition:

TOK_SETCOLOR	sizeof(int)	
Name Size	sizeof(int)	# characters in Color Name
Color Name	sizeof(char)*Name Size	Color Name string

Setmaterial Definition:

TOK_SETMATERIAL	sizeof(int)	
Name Size	sizeof(int)	# of characters in Material Name
Material Name	sizeof(char)*Name Size	Material Name String

Line Definition:

TOK_DEFLINE	sizeof(int)	
Number of Vertices	sizeof(int)	# of vertices
Vertices	sizeof(float)*3*Number of Vertices	Vertex Coordinates

Polygon Definition:

TOK_DEFPOLY	sizeof(int)	
Normal Array	sizeof(float)*3	Polygon Normal Coordinates
Number of Vertices	sizeof(int)	# of vertices
Vertices	sizeof(float)*3*Number of Vertices	Vertex Coordinates

Surface Definition:

TOK_DEFSURFACE	sizeof(int)	
Number of Vertices	sizeof(int)	# of vertices
Vertices	sizeof(float)*6*Number of Vertices	Vertex Coordinates followed by Vertex Normal Coordinates

APPENDIX C

UNDERSTANDING LEX

A. WHAT IS LEX

Lex is a program generator designed for lexical processing of character input streams. Lex is a tool available from within the UNIX operating system which generates a lexical analyzer module in the C programming language. The source for Lex is a high-level, problem oriented specification for character string matching, producing code which recognizes regular expressions from an input stream. These regular expressions are specified by the user in the source code provided to Lex [Ref. 7].

For a fairly complete and elementary introduction to Lex and its uses, refer to the publication "Lex - A Lexical Analyzer Generator" by M. E. Lesk and E. Schmidt. Researchers at NPS may find this instructive manual in the on-line help section of the VAX UNIX system of the Computer Science Department.

B. IMPORTANT FEATURES OF LEX

The lexical analyzer generated by Lex accepts ambiguous specifications and always matches the longest possible input at any point. The lexical analyzer performs substantial lookahead but does not recognize any regular expression longer than 200 characters. Hence, it is not possible to write a lexical analyzer with Lex that can recognize two different regular expressions if one is contained in the other.

The Lex routine essentially generates a finite state automaton which recognizes various regular expressions, or tokens. As each token is identified by the lexical analyzer the section of program code associated with that token, provided by the user, is

executed. Reading of the input stream is then resumed at the end of the last identified token.

C. THE USE OF LEX IN THE OFF

Because of the inherent capabilities of Lex, this tool was chosen to create a lexical analyzer capable of identifying the tokens in the OFF. The source code provided to Lex is found in the file **filelex**.

Lex is case sensitive, and for readability lower case implementation of all tokens was chosen. In most cases, tokens in the input stream merely cause an integer variable to be returned by the lexical analyzer. In some instances, such as the recognition of an identifier (string) or comment, the regular expression discovered by the lexical analyzer is copied to a global 'id' string for use in the OFF programs.

The **include** token causes suspension of input from the current file. A temporary storage structure is created, storing pertinent information on the current file. The included file is then opened for input and read until an end of file condition there or until another **include** is encountered. Once all input has been taken from the newest file, it is closed, and the former file is restored as the input stream using the temporarily stored information. Figures C.1 and C.2 show the relevant code from the file **filelex**.

D. ADDING NEW TOKENS TO THE OFF

Addition of new tokens to the OFF is a straightforward process, if the programmer understands the Lex tool and its implementation. Careful study of the reference document described in part A of this appendix is highly recommended.

Integers, alpha characters, alpha-numeric characters, comments, digits, floating point numbers, UNIX file names, identifiers, quoted strings, new line and white space have all been defined in Lex shorthand at the beginning of the **filelex** source code.

New tokens may incorporate these in their specification for concise and exact token specification.

Desired tokens may simply be inserted among the existing tokens in the **filelex** source code. The desired lexical analyzer program response should then be defined according to the Lex syntax. This can simply entail returning another integer token identifier or a more complicated program response.

Once all desired changes have been made to new or existing tokens, the module **filelex.c** must be regenerated for use in the OFF program routines. The UNIX call to generate this file has been included in the **makefile** in the OFF program directories. It is: *lex -t filelex > filelex.c.*

```
include() {

struct include_file   *temp;
char      filename[MAX_LINE];   /* file name from lex plus default path */
FILE    *f;

if (yylex()!=TOK_ID) {
  fprintf(stderr,">>> %s, line %d:\n",more_files->name,yylineno);
  fprintf(stderr,"    missing file name after include directive\n");
  return(0);
} else {
  if ((f = fopen(yytext,"r"))==NULL) {
    strcpy(filename,default_path);
    strcat(filename,yytext);
    if ((f = fopen(filename,"r"))==NULL) {
      fprintf(stderr,">>> %s : line %d :",more_files->name,yylineno);
      fprintf(stderr,"    unable to open file => %s\n",yytext);
      return(0);
    }
  }
  temp = (struct include_file *)malloc(sizeof(struct include_file));
  temp->line_count = yylineno;
  temp->name       = (char *)malloc((yyleng+1)*sizeof(char));
  temp->name = strncpy(temp->name,yytext,yyleng);
  temp->f    = f;
  yyin = f;
  temp->next = more_files;
  more_files = temp;
}  /* end if then else */
}  /* end include */
```

Figure C.1 Code Executed for File Include

```
yywrap() {

  struct include_file    *temp;

  if (more_files->next != NULL) {
    fclose( yyin );
    temp       = more_files;
    more_files = more_files->next;
    yyin       = more_files->f;
    yylineno   = more_files->line_count;
    free( temp );
    return(0);
  } else {
    fclose( yyin );
    end_of_input = TRUE;
    return(1);
  }
}
```

Figure C.2 Code Executed Upon End of File Condition

APPENDIX D

PREVIEW USERS MANUAL

A. AN OVERVIEW OF PREVIEW

The **preview** program is a 3D object viewing and manipulation program designed to interact with objects stored in an ASCII or binary version of the Object File Format (OFF). Its purpose is to allow the viewing and modification of objects stored in the OFF, thus permitting the user to interactively correct deficiencies in the object, verify specified object attributes (material definitions, lighting, etc.), and save any corrections made.

The C language program modules upon which **preview** is based are located in the *off/tools/preview* directory, as is the executable module, **preview**. This directory also includes a *makefile* for recompilation of the program module should future additions or corrections be made to the **preview** program.

B. HOW TO USE PREVIEW

Preview is invoked by entering the program name and providing the name of a 3D object file as an argument to the invocation, as in *preview (filename)*. For example, if the user wished to preview an aircraft carrier stored in the file carrier.nimitz, the invocation would appear as: *preview carrier.nimitz*.

Preview does not care what the name of the file is, provided it is a UNIX acceptable filename. **Preview** will determine if the file contains ASCII or binary data. If the file is an ASCII file, **preview** will attempt to read the file's data in the ASCII

OFF. If it is a binary file, **preview** will attempt to read the file's data in the binary OFF. If the file is neither an ASCII or binary file an error message will appear.

If the file specified in the call to **preview** is an ASCII or binary formatted file, but not a 3D object in the OFF, numerous error messages will appear as the data is read. If this occurs, it is a good idea to abort program execution by typing *control-C*. However, actual errors in the object file's data will also generate error messages. These should be noted so that the object file can be corrected.

The remainder of this appendix covers specific operations available within **preview** once the program is actually in operation.

C. THE PREVIEW WINDOWS

Once the 3D object has been read from the specified file, three graphics windows appear on the IRIS, completely covering the screen. The largest window, covering approximately the left two thirds of the screen from top to bottom, is referred to as the "viewport" window. The 3D object is displayed in this window at all times.

The other two windows appear in the right third of the screen, each taking approximately one half of the screen height. The top of these is referred to as the "tools" window and initially contains information telling where program options can be selected.

The lower of these two windows is the "help" window and initially contains program and object information. This information includes the number of lines, polygons and surfaces in the object, and the number of each currently being displayed; the specified object origin; the minimum and maximum x, y and z values of the object; the current resolution level; and instructions on the use of the left and middle mouse buttons. At all times in all windows, the right mouse button selects that window's currently active menu.

D. OTHER WINDOWS

The viewport window always remains the same, displaying the object as it is currently modified. The menu in that window likewise always remains the same.

A keyboard input window appears, contained entirely within the area of the viewport window, whenever an object data modification option has been selected. When it appears, it is the active window and does not disappear until keyboard input has been completed. As soon as keyboard input has been completed, as discussed later, this window disappears and the viewport window is again completely visible.

The tools and help windows change their display in response to options selected by the user. At all times, the help window provides information pertinent to the operation selected and information on the current operations performed by the left and middle mouse buttons. The only option ever available in any help window menu is "Exit Program."

The tools window either contains its main display or contains one, two or three slider bars. The operations performed by the slider bars are always outlined in the help window. The left mouse, held down, allows the cursor to move the slider button over which it, the cursor, is positioned. Various menu options are available, depending upon which sliders are currently displayed in the tools window. However, the "Exit Slider" option is always contained in the tools window menu when slider bars are present.

E. SELECTING PREVIEW OPTIONS

As stated above, only the "Exit Program" option can be selected from the help window menu. The viewport window menu always contains the same options. These are shown in Figure D.1.

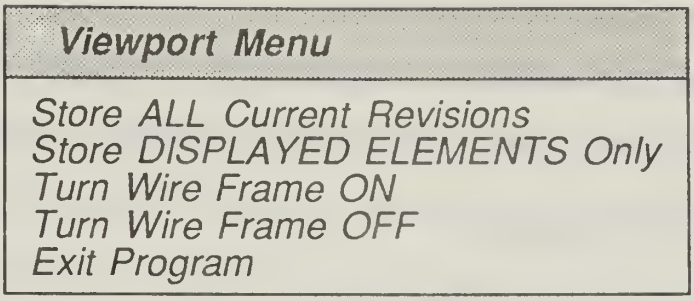

Figure D.1 Viewport Menu Options

The majority of options are selected from the main tools window menu. These are shown in Figure D.2.

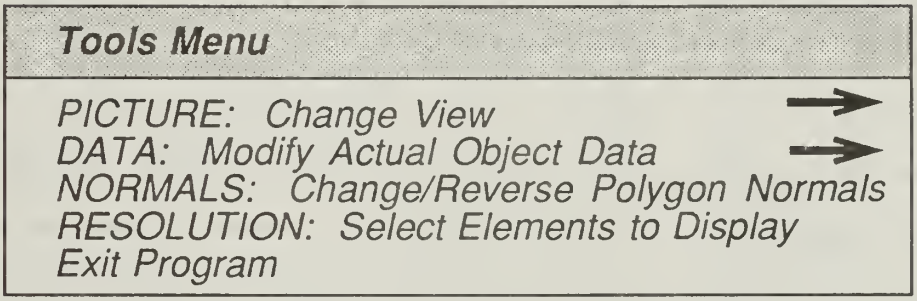

Figure D.2 Main Tools Window Menu

Menu options are selected by pressing and holding the right mouse button when the cursor is in the window where the desired menu resides. Then, once the menu appears, the desired operation on the menu is highlighted by moving the mouse while holding the right mouse button down. Finally, the option is selected by releasing the right mouse when the desired option is highlighted.

Two menu options in the main tools window menu have "roll-off" sub-menus. These are indicated by a right arrow following the menu choice. This indicates the need to "refine" the choice by selecting further options under the main menu option. To do so, highlight the menu option required, then move the mouse to the right. Another menu will appear. It may also contain a sub-menu, again indicated by a right

arrow. Continue this process until the final sub-menu appears, highlight the desired option on it, then release the right mouse button.

Upon accidental menu activation, simply move the cursor until nothing is highlighted, then release the mouse button. This results in a "no choice" situation.

F. THE PREVIEW OPTIONS

1. Changing The View of the Object

The object, or "picture" seen in the viewport window, can be rotated or translated to provide a different viewing aspect or perspective. This option is the first available choice in the main tools window menu. This option provides a roll off sub-menu from which rotation or translation can be selected.

Once translation or rotation is selected, three slider bars appear in the tools window, one for each axis. Moving each slider rotates or translates the object about or along the indicated axis. The object moves in the viewport window as the slider bar is moved. If the slider bar gets to one end of its span, the sliders can be "reset" to the middle by selecting the "Reset Slider to Middle" option from the current tools window menu.

Continue rotation and or translation of the object until the desired "picture" is observed in the viewport window. Then, choose the "Exit Slider" option from the current tools window menu to return to the main level. Figure D.3 shows the slider bars menu available in the tools window when these sliders are active.

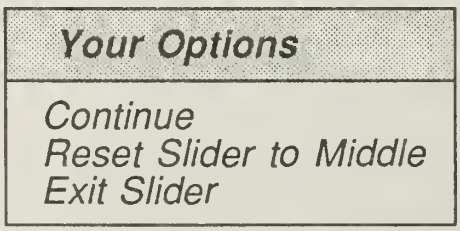

Figure D.3 Slider Bar Menu

2. Object Data Modification

The object data modification option is one of the options at the heart of the **preview** program. It allows actual modification of the data from the object file. Such modification results in actual transformation of the object data stored in the dynamic structure created by **preview**. Saving the file after such transformations creates a new file with this modified object data.

This option is also selected from the main tools window menu. It consists of two sub-levels of roll-off menus. The first level lets the user choose whether modification is to be by rotation, scaling or translation. The second level then selects which axis the modification should take place about or around. Only scaling can be performed about all axes simultaneously.

Once the desired modification process has been selected, the keyboard input window appears in the center of the viewport window. Keyboard input is now sought, and an appropriate prompt appears in this new window. Only the minus (-) key and the numeric keys above the standard keyboard are active, along with the delete key and the return key. Enter the desired modification amounts, using the delete key to correct errors, then press return when the input is correct. The keyboard input window disappears, the object is modified, and the information displayed in the main help window reflects the new values.

One important note needs to be made. The object is always displayed with its center in the middle of the viewport window, unless translated by the "picture" slider bars discussed previously. Hence, even if a modification by translation is performed, the view in the viewport appears unchanged. However, note the new minimum and maximum values in the main help window, and the new object origin there as well.

If the user desires to save the new object data to a new file, he should be aware of one important potential problem. If the object file relied on the default origin of (0.0, 0.0, 0.0) by failing to specify an object origin, the new file will likewise have no specified origin despite the fact that a modification by translation creates a new object origin. If such a modification is anticipated, the object file should contain a specified origin, even if that origin is the standard (0.0, 0.0, 0.0). That way, should the object's origin be modified by a translation, the new file will contain the modified origin when it is saved. The other alternative is to remember what the new origin should be. Then add it to the saved file once **preview** has been exited.

3. Reversing Polygon Normals

The "NORMALS: Change/Reverse Polygon Normals" option in the main tools window menu causes one slider bar to appear in the tools window. This slider bar is then used to select polygons in the object.

The current polygon is always highlighted in white, with a red border line drawn around its perimeter. The slider bar starts with the number one at the bottom, and the number of total polygons in the object at the top. As the slider is moved, the current, highlighted polygon moves throughout the object. The slider bar is intended for quick access and movement. However, when there are large numbers of polygons in the object, it can be difficult or impossible to precisely position the slider bar to select the desired polygon. Figure D.4 shows the menu available in the tools window when this slider is active.

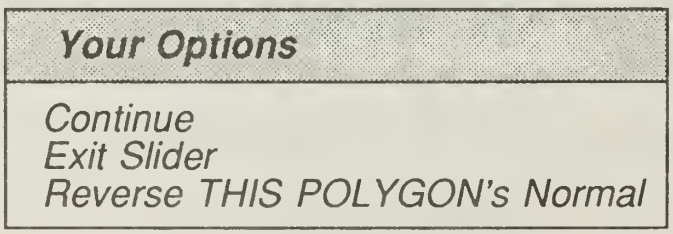

Figure D.4 Normal's Slider Bar Menu

At all times during this slider operation, the middle mouse button increments the slider button by one. Therefore, use the slider to select a current polygon several polygons "below" the desired polygon. Then fine tune the precise selection with the middle mouse button. Notice that the number of the current polygon is always displayed on the slider button itself.

Once the desired polygon is highlighted, use the menu now available in the tools window to "Reverse THIS POLYGON'S Normal". Notice that the polygon data displayed in the help window shows the normal being reversed. Continue selecting and reversing normals in a like manner. This slider can be exited by the "Exit Slider" option in the tools window menu. The "picture" can then be rotated or translated to see another aspect of the object, and more normal corrections made.

4. Object Resolution

When at the main level, use of the left and middle mouse buttons effect the resolution level, as outlined in the help window. This permits quick, rough approximations of various resolution levels. Use these buttons to obtain a resolution as close to the desired level as possible. After such a level is obtained, fine tuning can be accomplished with the "RESOLUTION: Select Elements to Display" option in the main tools window menu.

Once this option has been selected, two slider bars appear in the tools window. They are similar in function to the normal's slider bar discussed in the previous section. In fact, the left slider bar is for selecting a current polygon, and the right is used to select a current surface. If either surfaces or polygons are not present in the object, a zero (0) appears at the top of the appropriate slider, and the slider is not moveable.

Use these sliders to choose the surface or polygon to add or delete from the present resolution level of the object. The current polygon is again highlighted in white, the current surface is highlighted in yellow. Each has a red border around its perimeter. Note that only the most recently moved slider bar has its current element highlighted. That is, if the polygon slider is moved, the current polygon is highlighted. If the surface slider is moved, the current surface is highlighted. Use of the middle mouse while these sliders are active increments both sliders simultaneously by one. Figure D.5 shows the menu available in the tools window when these sliders are active.

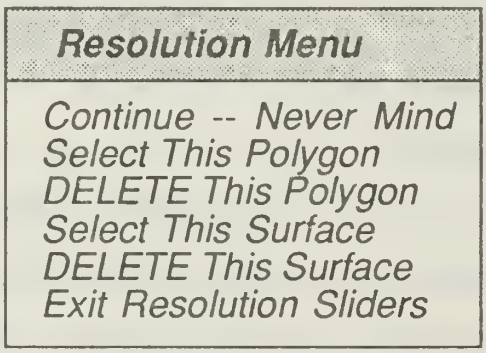

Figure D.5 Resolution Slider Bars Menu

Once the desired polygon or surface has been highlighted it can be added to the current resolution or deleted from it. These options are contained in the current tools window menu. Continue to highlight and select or delete polygons and/or surfaces in a like manner. The sliders can be exited by the "Exit Resolution Sliders" option in the current tools window menu. Rotate and translate the object picture to see various aspects, and continue to select or delete elements until the desired resolution is obtained.

G. CREATING NEW OBJECT FILES

New object files are created by the save options available in the viewport window menu. The user can save either all object data, which should be done in the case of object modification, or save only the currently displayed elements, in order to preserve a new resolution of the object.

The new file created by **preview** is always in the same format, ASCII or binary, as the input file for the object. The file name of the new file is the same as the input file, with the tag ".new" appended to it.

Saving can be selected at any time when using **preview**, and can be selected repeatedly. Each save writes to the same file, so any mistakes can simply be corrected and overwritten with no difficulty. However, should the user desire to make several new files of various resolutions from one use of **preview**, the IRIS's side terminal should be used to move the new file after each save before any additional modification and saving occurs.

H. WIRE FRAME MODE

The final feature of **preview** is the wire frame option. As the name implies, this displays the object in a "skeleton" or "wire frame" configuration, where each polygon or surface is drawn as an outline instead of a solid piece. This option is selected from the viewport window menu and can be selected at any time.

This option can be useful for determining where problems lie in connecting object elements or in seeing where "holes" occur at a lower resolution level without having to exit the resolution sliders. Additionally, it provides an interesting way to study the complexity of an object composed of many hundreds of polygons.

The outline, or wire frame, of each polygon and surface is drawn in a color determined by the material characteristics that the polygon or surface would otherwise

use in normal viewing. For lines, the color is the defined and set color as appears in the normally viewed object.

LIST OF REFERENCES

1. Smith, Douglas B., and Streyle, Dale G., *An Inexpensive Real-Time Interactive Three-Dimensional Flight Simulation System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

2. Oliver, Michael R., and Stahl, David J., *Interactive, Networked, Moving Platform Simulators*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987.

3. Adams, Rodney M., *A Software Architecture for a Commander's Display System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, April 1987.

4. Harris, Frank E., *Preliminary Work on the Command and Control Workstation of the Future,* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1988.

5. Phillips, Charles E. Jr., and Weeks, Gordon K. Jr., *Command and Control Workstation of the Future Subsurface and Periscope Views,* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1989.

6. Silicon Graphics Inc., *IRIS Users's Guide,* v. 1, Mountain View, California, 1987.

7. Lesk, M. E. and Schmidt, E., *Lex -- A Lexical Analyzer Generator,* VAX/UNIX System Manual, Naval Postgraduate School, Monterey, California, 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Michael J. Zyda Naval Postgraduate School Code 52, Department of Computer Science Monterey, CA 93943-5100	200
4.	LCDR John M. Yurchak Naval Postgraduate School Code 52, Department of Computer Science Monterey, CA 93943-5100	1
5.	LT Steven A. Munson U. S. Coast Guard Supply Center Curtis Bay Baltimore, MD 21226-1792	2
6.	John Maynard Naval Ocean Systems Center Code 402 San Diego, CA 92152	1
7.	Duane Gomez Naval Ocean Systems Center Code 433 San Diego, CA 92152	1
8.	James R. Louder, Naval Underwater Systems Center Combat Control Systems Department Building 1171/1, Newport, RI 02841	1

9. Superintendent
Attn: Research Administration, Code 012
Naval Postgraduate School
Monterey, CA 93943-5000 1

Thesis
M9544 Munson
c.1 Integrated support for manipulation and display of 3D objects for the Command and Control Work-station of the Future.

Thesis
M9544 Munson
c.1 Integrated support for manipulation and display of 3D objects for the Command and Control Work-station of the Future.

thesM9544
Integrated support for manipulation and
3 2768 000 83061 6
DUDLEY KNOX LIBRARY